/burl@stx null def /BU.S /burl@stx null def def /BU.SS currentpoint /burl@lly exch def /burl@llx exch def bu

The Beginner's Textbook for Fully Homomorphic Encryption

Ronny Ko *
LG Electronics Inc.

* *Acknowledgments:*
Robin Geelen (KU Leuven)
Tianjian Yang (Peking University)
Yongwoo Lee (Inha University)
Nolan Carouge (Grenoble INP Ensimag)

Preface

Fully Homomorphic Encryption (FHE) is a cryptographic scheme that enables computations to be performed directly on encrypted data, as if the data were in plaintext. After all computations are performed on the encrypted data, it can be decrypted to reveal the result. The decrypted value matches the result that would have been obtained if the same computations had been applied to the plaintext data.

FHE supports basic operations such as addition and multiplication on encrypted numbers. Using these fundamental operations, more complex computations can be constructed, including subtraction, division, logic gates (e.g., AND, OR, XOR, NAND, MUX), and even advanced mathematical functions such as ReLU, sigmoid, and trigonometric functions (e.g., sin, cos). These functions can be implemented either as exact formulas or as approximations, depending on the trade-off between computational efficiency and accuracy.

FHE enables privacy-preserving machine learning by allowing a server to process the client' s data in its encrypted form through an ML model. With FHE, the server learns neither the plaintext version of the input features nor the inference results. Only the client, using their secret key, can decrypt and access the results at the end of the service protocol. FHE can also be applied to confidential blockchain services, ensuring that sensitive data in smart contracts remains encrypted and confidential while maintaining the transparency and integrity of the execution process. Other applications of FHE include secure outsourcing of data analytics, encrypted database queries, privacy-preserving searches, efficient multi-party computation for digital signatures, and more.

This book is designed to help the reader understand how FHE works at the mathematical level. The book comprises the following four parts:

- **??: ??** explains necessary background concepts for FHE, such as groups, fields, orders, polynomial rings, cyclotomic polynomials, vectors and matrices, the Chinese Remainder Theorem, Taylor series, polynomial interpolation, and the Fast Fourier Transform.
- **??: ??** explains well-known lattice-based cryptographic schemes, which include LWE, RLWE, GLWE, GLev, and GGSW cryptosystems.
- **??: ??** explains the generic techniques of FHE adopted by many existing schemes, such as homomorphic addition, multiplication, modulus switching, and key switching.
- **??: ??** explains four widely used FHE schemes: TFHE, BFV, CKKS, and BGV, as well as their RNS-variant versions.

This book is available both as an **arXiv PDF** and on an **auto-generated website** (powered by make4ht). Please report any bugs or suggestions regarding the draft to the **Issues Board**. As this book is an evolving open project, we welcome FHE experts to join us as collaborators and help expand the draft.

Acknowledgments

Special thanks go to the following researchers: Robin Geelen (KU Leuven, robin.geelen@esat.kuleuven.be), for his thoughtful and dedicated feedback; Yongwoo Lee (Inha University, yongwoo@inha.ac.kr), for his general advice; Tianjian Yang (Peking University, 2300012738@stu.pku.edu.cn), for correcting

numerous typos and other errors in the draft; and Nolan Carouge (Grenoble INP Ensimag,), for further strengthening the draft' s theoretical exposition.

Contents

Part I
Basic Math

This chapter explains the basic mathematical components of number theory: group, field, order, roots of unity, cyclotomic polynomial, polynomial ring, and decomposition. These are essential building blocks for post-quantum cryptography.

A-1 Modulo Arithmetic

- Reference

YouTube – Extended Euclidean Algorithm Tutorial.

A-1.1 Overview

⟨Definition ??⟩ Integer Modulo

- **Modulo** is the operation of computing the remainder obtained when one number is divided by another. **modulo** is often abbreviated as **mod**.
- ***a* mod *q* (i.e., *a* modulo *q*)** is the remainder after dividing a by q, which is always an element of $\{0, 1, 2, 3, \cdots, q-1\}$. For example, $7 \bmod 5 = 2$, because the remainder of dividing 7 by 5 is 2.
- **Modulus:** Given $\boldsymbol{a}$ mod $\boldsymbol{q}$, we call the divisor q the modulus, whereas *modulo* refers to the operation.
- **Modulo Congruence (≡):** a is congruent to b modulo q (i.e., $a \equiv b$ **mod** q) if they have the same remainder when divided by q. For example, $5 \equiv 12 \bmod 7$, because $5 \bmod 7 = 5$ and $12 \bmod 7 = 5$. In mathematics, the notation $a \equiv b \bmod q$ is identical to $a = b \pmod q$, meaning that the remainder of a divided by q is the same as the remainder of b divided by q. Note that this notation differs from $a = b \bmod q$, which states that a equals the remainder of b divided by q.
- **Congruence *vs.* Equality:**
  $a \equiv b \bmod q \iff a = b + k \cdot q \quad$ (for some integer k)

  This means that a and b are congruent modulo q if and only if a and b differ by some multiple of q. For example, $5 \equiv 12 \bmod 7 \iff 5 = 12 + (-1) \cdot 7$

A-1.2 Modulo Arithmetic

The supported modulo operations are addition, subtraction, and multiplication. The properties of these modulo operations are as follows:

⟨Theorem ??.1⟩ Properties of Modulo Operations

For any integer x, the following is true:

1. **Addition:** $a \equiv b \bmod q \iff a + x \equiv b + x \bmod q$
2. **Subtraction:** $a \equiv b \bmod q \iff a - x \equiv b - x \bmod q$
3. **Multiplication:** $a \equiv b \bmod q \iff a \cdot x \equiv b \cdot x \bmod q$. This equivalence holds provided that $\gcd(x, q) = 1$. Without this assumption, only the implication $a \equiv b \bmod q \Rightarrow a \cdot x \equiv$

$b \cdot x \bmod q$ is guaranteed.

Proof.

For any integer x,

1. **Addition:** $a \equiv b \bmod q \iff a = b + kq$ (for some integer k) # a and b differ by some multiple of q
   $\iff a + x = b + k \cdot q + x$
   $\iff a + x = b + x + k \cdot q$ # $a + x$ and $b + x$ differ by some multiple of q
   $\iff a + x \equiv b + x \bmod q$

2. **Subtraction:** $a \equiv b \bmod q \iff a = b + kq$ (for some integer k)
   $\iff a - x = b + k \cdot q - x$
   $\iff a - x = b - x + k \cdot q$ # $a - x$ and $b - x$ differ by some multiple of q
   $\iff a - x \equiv b - x \bmod q$

3. **Multiplication:** $a \equiv b \bmod q \iff a = b + kq$ (for some integer k)
   $\implies a \cdot x = b \cdot x + k \cdot q \cdot x$
   $\implies a \cdot x = b \cdot x + k_x \cdot q$ (where $k_x = k \cdot x$) # $a \cdot x$ and $b \cdot x$ differ by some multiple of q
   $\implies a \cdot x \equiv b \cdot x \bmod q$
   Conversely, if x and q are coprime (i.e., $\gcd(x, q) = 1$), then x has a multiplicative inverse x^{-1} modulo q. From $a \cdot x \equiv b \cdot x \bmod q$
   $\implies a \cdot x \cdot x^{-1} \equiv b \cdot x \cdot x^{-1} \bmod q$
   $\implies a \equiv b \bmod q$

□

Based on the modulo operations in Theorem **??**.1, we can also derive the following properties of modulo arithmetic:

⟨Theorem ??.2⟩ Properties of Modulo Arithmetic

1. **Associative:** $(a \cdot b) \cdot c \equiv a \cdot (b \cdot c) \bmod q$
2. **Commutative:** $(a \cdot b) \equiv (b \cdot a) \bmod q$
3. **Distributive:** $(a \cdot (b + c)) \equiv ((a \cdot b) + (a \cdot c)) \bmod q$
4. **Interchangeable:** Congruent values are interchangeable in modulo arithmetic. For example, suppose $(a \equiv b \bmod q)$ and $(c \equiv d \bmod q)$. Then, a and b are interchangeable, and c and d are interchangeable in modulo arithmetic as follows:
   $(a + c) \equiv (b + d) \equiv (a + d) \equiv (b + c) \bmod q$
   $(a - c) \equiv (b - d) \equiv (a - d) \equiv (b - c) \bmod q$
   $(a \cdot c) \equiv (b \cdot d) \equiv (a \cdot d) \equiv (b \cdot c) \bmod q$

The proof of Theorem **??**.2 is similar to that of Theorem **??**.1, which we leave as an exercise for the reader.

A-1.3 Inverse

⟨Definition ??⟩ Inverse in Modulo Arithmetic

In modulo q (i.e., in the world of remainders where all numbers have been divided by q), for each $a \in \{0, 1, 2, \cdots, q-1\}$:

- **Additive Inverse** of a is denoted as a_+^{-1} which satisfies $a + a_+^{-1} \equiv 0 \bmod q$. For example, in modulo 11, $3_+^{-1} = 8$, because $3 + 8 \equiv 0 \bmod 11$.
- **Multiplicative Inverse** of a is denoted as a_*^{-1} which satisfies $a \cdot a_*^{-1} \equiv 1 \bmod q$. Such an inverse exists if and only if $\gcd(a, q) = 1$. For example, modulo 11, $3_*^{-1} = 4$, because $3 \cdot 4 \equiv 1 \bmod 11$.

A-1.4 Modulo Division

In modulo arithmetic, *modulo division* is different from regular numeric division. Strictly speaking, there is no such thing as *modulo division* because the modulo operation itself is a form of division that returns only the remainder. In practice, one uses "modulo division" to mean multiplying by a modular inverse when it exists, i.e., when $\gcd(a, q) = 1$. *Modulo division* of b by a modulo q is equivalent to computing the *modular* multiplication $b \cdot a^{-1} \bmod q$. The result of *modulo division* is different from that of numeric division because *modulo division* always gives an integer (a residue modulo q) (as it multiplies two integers modulo q), whereas numeric division gives a real number. The inverse of an integer modulo q can be computed using the extended Euclidean algorithm (YouTube tutorial).

A-1.5 Centered Residue Representation

Throughout this section, we have assumed that the residues are positive integers. For example, the possible residues modulo q are assumed to be $\{0, 1, \cdots, q-1\}$. This system is called the canonical (i.e., unsigned) residue representation. On the other hand, there is also a counterpart system that assumes signed (i.e., centered) residues $\left\{-\frac{q}{2}, -\frac{q}{2}+1, \cdots, 0, \cdots, \frac{q}{2}-2, \frac{q}{2}-1\right\}$[1], with the residues centered around 0, and the total number of residues is the same, namely q. In both systems, a modulo operation changes a given value to another value within the system's residue range such that: (1) if the given value is greater than the upper bound of the residue range, the value is subtracted by the modulus q; (2) if the value is less than the lower bound of the residue range, the value is increased by the modulus q. The only difference between these two (canonical and centered) systems is their upper bounds and lower bounds: 0 and $q-1$ in the canonical residue system, whereas $-\frac{q}{2}$ and $\frac{q}{2}-1$ in the centered residue system. The canonical residue representation assumes that $\mathbb{Z}_q = \{0, 1, \cdots, q-1\}$, whereas the centered residue system assumes that $\mathbb{Z}_q = \left\{-\frac{q}{2}, -\frac{q}{2}+1, \cdots, 0, \cdots, \frac{q}{2}-2, \frac{q}{2}-1\right\}$.

In both systems, the same modulo property of addition, subtraction, multiplication, and division holds, which can be proved by applying the same reasoning described in **??**: the same properties

[1] Here, we assume q is an even number. In the case where q is an odd number, the residues are $\left\{-\frac{q-1}{2}, -\frac{q-3}{2}, \cdots, 0, \cdots, \frac{q-3}{2}, \frac{q-1}{2}\right\}$

hold in both systems because any two congruent residues in the centered system are separated by the kq gaps (for some integer k) in both systems.

Also, the same property holds for an inverse: an inverse of a modulo q is a^{-1} such that $a \cdot a^{-1} \equiv 1 \bmod q$.

Using a signed residue representation is useful in certain cases. In an example of canonical (i.e., unsigned) residue representation, suppose we have the relation $a + b \bmod q$ and we know that in a given application, $a+b$ is guaranteed to be within the $[0, q-1]$ range (i.e., $0 \leq a+b \leq q-1$). Then, $(a + b \bmod q) = a + b$, and thus we can remove the modulo operation, simplifying the relation. Now, suppose a different example of centered (i.e., signed) residue representation where we have the relation $a - b \bmod q$, and we know that in a given application, $a - b$ is guaranteed to be within the range $\left[-\frac{q}{2}, \frac{q}{2} - 1\right]$. Then, $(a - b \bmod q) = a - b$. However, notice that if the relation $a - b \bmod q$ were in a canonical residue representation, then we cannot remove the modulo operation because if $a - b$ is negative, then this becomes smaller than the lower bound of the canonical residue system (i.e., 0), and thus a modulo reduction (i.e., addition by one or more q) is needed.

In **??**, we design the FastBConvEx operation based on this beneficial property of centered residue representation: in this algorithm design, we can simplify $(\mu + u \bmod b_\alpha)$ to $\mu + u$ because we know that $-\frac{b_\alpha}{2} \leq \mu + u < \frac{b_\alpha}{2}$.

A-2 Group

A-2.1 Definitions

⟨Definition ??⟩ Group

Set Elements

- **Set ($\mathbb{S}$):** A bundle of elements: $\mathbb{S} = \{a, b, c, \ldots\}$
- **Set Operations ($+, \cdot$):** We consider two operations on $\mathbb{S}$ between any two elements $a, b \in \mathbb{S}$ as operands: addition ($+$) and multiplication ($\cdot$)
- **Additive Identity ($0_{(+)}$ often written 0):** An element $i \in \mathbb{S}$ is an additive identity if for all $a \in \mathbb{S}$, $i + a = a$.
- **Multiplicative Identity ($1_{(\cdot)}$ often written 1):** An element $i \in \mathbb{S}$ is a multiplicative identity if for all $a \in \mathbb{S}$, $i \cdot a = a$
- **Additive Inverse ($a^{-1}_{(+)}$):** For each $a \in \mathbb{S}$, its additive inverse $a^{-1}_{(+)}$, often written $-a$, is defined as an element such that $a + a^{-1}_{(+)} = 0_{(+)}$ (i.e., additive identity)
- **Multiplicative Inverse ($a^{-1}_{(\cdot)}$):** For each $a \in \mathbb{S}$ that is invertible with respect to $(\cdot)$, its multiplicative inverse $a^{-1}_{(\cdot)}$, often written a^{-1}, is defined as an element such that $a \cdot a^{-1}_{(\cdot)} = 1_{(\cdot)}$ (i.e., multiplicative identity)

Element Operation Features

- **Closed:** A set $\mathbb{S}$ is closed under the $(+)$ operation if for every $a, b \in \mathbb{S}$, it is the case that $a + b \in \mathbb{S}$. Likewise, a set $\mathbb{S}$ is closed under the $(\cdot)$ operation if for every $a, b \in \mathbb{S}$, it is the case that $a \cdot b \in \mathbb{S}$.
- **Associative:** $(a + b) + c = a + (b + c)$
- **Commutative:** $a + b = b + a$
- **Distributive:** When both $(+)$ and $(\cdot)$ are defined, $a \cdot (b + c) = (a \cdot b) + (a \cdot c)$

Group Types

- **Semigroup:** A semigroup is a set of elements which is closed and associative on a single operation ($+$ or $\cdot$)
- **Monoid:** A monoid is a semigroup, plus it has an identity element e, which returns the other operand over the set operation.
  (e.g., 0 is the identity element for $+$ operator, 1 is the identity element for the $\cdot$ operator)
- **Group:** A group is a monoid, and every element has an inverse with respect to the operation.
- **Abelian Group:** An abelian group is a group, plus its operation is commutative.

A-2.2 Examples

$\mathbb{Z}$ (i.e., the set of all integers) is an abelian group under addition $(+)$, because:

- **Closed:** For any integer $a, b \in \mathbb{Z}$, $a + b = c$ is also an integer $(\in \mathbb{Z})$.
- **Associative:** For any integer $a, b, c \in \mathbb{Z}$, $(a + b) + c = a + (b + c)$.
- **Identity:** The additive identity is 0 because, for any $a \in \mathbb{Z}$, $a + 0 = a$.

- **Inverse:** For each $a \in \mathbb{Z}$, its additive inverse is $-a$, as $a + (-a) = 0$.
- **Commutative:** For any integer $a, b \in \mathbb{Z}$, $a + b = b + a$.

$\mathbb{Z}$ is a monoid under multiplication $(\cdot)$ because:

- **Closed:** For any integer $a, b \in \mathbb{Z}$, $a \cdot b = c$ is also an integer $(\in \mathbb{Z})$.
- **Associative:** For any integer $a, b, c \in \mathbb{Z}$, $(a \cdot b) \cdot c = a \cdot (b \cdot c)$.
- **Identity:** The multiplicative identity is 1, because for any $a \in \mathbb{Z}$, $a \cdot 1 = a$.
- **NO Inverse:** For an integer $a \in \mathbb{Z}$, its multiplicative inverse is $\frac{1}{a}$, but this is not necessarily an integer $(\notin \mathbb{Z})$; therefore, not every element has a multiplicative inverse. Thus, $(\mathbb{Z}, \cdot)$ is not a group (though it is a monoid).

$\mathbb{R}^\times$ (i.e., the set of all nonzero real numbers) is an abelian group under multiplication $(\cdot)$, because:

- **Closed:** For any real number $a, b \in \mathbb{R}^\times$, $a \cdot b = c$ is also a real number $(\in \mathbb{R}^\times)$.
- **Associative:** For any real number $a, b, c \in \mathbb{R}^\times$, $(a \cdot b) \cdot c = a \cdot (b \cdot c)$.
- **Identity:** The multiplicative identity is 1, as for any real number $a \in \mathbb{R}^\times$, $a \cdot 1 = a$.
- **Inverse:** For each real number $a \in \mathbb{R}^\times$, its multiplicative inverse is $\frac{1}{a}$, which is a real number $(\in \mathbb{R})$.

A-3 Field

- Reference: Fields and Cyclotomic Polynomials [**?**]

A-3.1 Definitions

⟨Definition ??⟩ Field Definitions

- **Ring:** A set R that is an abelian group under addition $(+)$, equipped with a multiplication $(\cdot)$ that is closed and associative, and such that multiplication distributes over addition on both sides: $a\cdot(b+c) = a\cdot b+a\cdot c$ and $(a+b)\cdot c = a\cdot c+b\cdot c$ for all $a, b, c \in R$. (*Multiplication is not necessarily commutative* (*e.g., a matrix multiplication*)*, and an identity element for* $(\cdot)$ *is optional unless stated "ring with unity".*)
- **Field:** A set F that is an abelian group under $(+)$, whose nonzero elements $F^\times = F \setminus \{0\}$ form an abelian group under $(\cdot)$, with multiplication distributing over addition.
- **Galois Field** ($\mathrm{GF}(p^n)$)**:** A field with a finite number of elements, necessarily p^n for some prime p and positive integer n.
- $\mathbb{Z}_p$ ($\mathbb{Z}/p\mathbb{Z}$)**:** For a prime p, the set $\{0, 1, \ldots, p-1\}$ with addition and multiplication modulo p forms a finite field. More generally, for any integer $m \geq 2$, $\mathbb{Z}_m$ is a commutative ring, and it is a field iff m is prime.

A-3.2 Examples

$\mathbb{Z}$ (the set of all integers) is a ring but not a field, because not all of its elements have a multiplicative inverse (as shown in **??**).

$\mathbb{R}$ (the set of all real numbers) is a field. As shown in **??**, it is an abelian group under $(+)$; its nonzero elements form an abelian group under $(\cdot)$, and multiplication distributes over addition.

$\mathbb{Z}_7 = \{0, 1, 2, 3, 4, 5, 6\}$ is a finite field because:

- **Closed:** For any $a, b \in \mathbb{Z}_7$, there exist $c_1, c_2 \in \mathbb{Z}_7$ such that $a + b \equiv c_1 \pmod 7$ and $a \cdot b \equiv c_2 \pmod 7$.
- **Associative:** For any $a, b, c \in \mathbb{Z}_7$, $(a + b) + c = a + (b + c)$, and $(a \cdot b) \cdot c = a \cdot (b \cdot c)$.
- **Commutative:** For any $a, b \in \mathbb{Z}_7$, $a + b = b + a$, and $a \cdot b = b \cdot a$.
- **Distributive:** For any $a, b, c \in \mathbb{Z}_7$, $(a + b) \cdot c = a \cdot c + b \cdot c$.
- **Identity:** The additive identity is 0, and the multiplicative identity is 1.
- **Inverse:** For any $a \in \mathbb{Z}_7$, there exists $a' \in \mathbb{Z}_7$ such that $a + a' \equiv 0 \pmod 7$ (e.g., the additive inverse of 3 is 4 since $3 + 4 \equiv 0 \pmod 7$). For any $a \in \mathbb{Z}_7^\times = \{1, \ldots, 6\}$, there exists $b \in \mathbb{Z}_7$ such that $ab \equiv 1 \pmod 7$ (e.g., $3 \cdot 5 = 15 \equiv 1 \pmod 7$).

A-3.3 Theorems

⟨Theorem ??⟩ Field Theorems

1. **Size of Finite Field:** Every finite field is called a Galois Field and it has p^n elements for some prime p and positive integer n, conversely, for each p^n there exists a finite field of that size (unique up to isomorphism).
2. **Isomorphic Fields:** Any two finite fields $\mathbb{F}_1$ and $\mathbb{F}_2$ with the same number of elements are isomorphic, i.e., there exists a bijection $f : \mathbb{F}_1 \to \mathbb{F}_2$ such that for all $a, b \in \mathbb{F}_1$, $f(a+b) = f(a) + f(b)$ and $f(ab) = f(a)f(b)$.

A-4 Order

- Reference: Fields and Cyclotomic Polynomials [**?**]

A-4.1 Definitions

⟨Definition ??⟩ Order Definition

$\mathsf{ord}_{\mathbb{F}}(a)$**:** For $a \in \mathbb{F}^{\times}$ (a finite field, **??**), a's order is the smallest positive integer k such that $a^k = 1$.

A-4.2 Theorems

⟨Theorem ??.1⟩ Order Property (I)

For $a \in \mathbb{F}^{\times}$, and $n \geq 1$, $a^n = 1$ if and only if $\mathbf{ord}_{\mathbb{F}}(a) \mid n$
(i.e., $\mathsf{ord}_{\mathbb{F}}(a)$ divides n).

Proof.

1. *Forward Proof:* If $\mathsf{ord}_{\mathbb{F}}(a) \mid n$, then for $\mathsf{ord}_{\mathbb{F}}(a) = k$ where k is a's order, and $n = lk$ for some integer l.
   Then, $a^n = a^{lk} = (a^k)^l = 1^l = 1$.
2. *Backward Proof:* If $a^n = 1$ and $\mathsf{ord}_{\mathbb{F}}(a) = k$, write $n = qk + r$ with $0 \leq r < k$. Then $1 = a^n = a^{qk+r} = (a^k)^q a^r = a^r$. By minimality of k, we must have $r = 0$, hence $k \mid n$.

□

⟨Theorem ??.2⟩ Order Property (II)

If $\mathsf{ord}_{\mathbb{F}}(a) = k$, then for any $n \geq 1$, $\mathsf{ord}_{\mathbb{F}}(a^n) = \dfrac{k}{\gcd(k, n)}$.

Proof.

1. $a^k, a^{2k}, a^{3k}, \ldots = 1$.
2. Given $\mathsf{ord}_{\mathbb{F}}(a^n) = m$, $(a^n)^m, (a^n)^{2m}, (a^n)^{3m}, \ldots = 1$
3. Note that by definition of order, $x = k$ is the smallest value that satisfies $a^x = 1$. Thus, given $\mathsf{ord}_{\mathbb{F}}(a^n) = m$, then m is the smallest integer that makes $(a^n)^m = 1$. Note that $(a^n)^m$ should be a multiple of a^k, which means mn should be a multiple of k. The smallest possible integer m that makes mn a multiple of k is $m = \dfrac{k}{\gcd(k, n)}$.

□

⟨Theorem ??.3⟩ Order Property (III)

$\mathsf{ord}_{\mathbb{F}}(a) = kn$ if and only if $\mathsf{ord}_{\mathbb{F}}(a^k) = n$.

Proof.

1. *Forward Proof:* Given $\mathsf{ord}_{\mathbb{F}}(a) = kn$, and given Theorem **??**.2, $\mathsf{ord}_{\mathbb{F}}(a^k) = \dfrac{nk}{\gcd(k, nk)} = \dfrac{nk}{k} = n$.
2. *Backward Proof:* Given $\mathsf{ord}_{\mathbb{F}}(a^k) = n$ and letting $\mathsf{ord}_{\mathbb{F}}(a) = m$, Theorem **??**.2 gives $m/\gcd(m, k) = n$, so $m = n\gcd(m, k)$. In particular $k \mid m$, hence $m = nk$.

□

⟨Theorem ??.4⟩ Fermat's Little Theorem

Given $|\mathbb{F}| = p$ (a prime) and $a \in \mathbb{F}$, $a^p = a$.

Proof.

1. If $a = 0$, then $a^p = a = 0$.
2. If $a \neq 0$, then $a \in \mathbb{F}^\times$, the multiplicative group of the field, which has size $|\mathbb{F}^\times| = p - 1$. By Lagrange's theorem (in a finite group G, the order of any element divides $|G|$), the order of a divides $p - 1$, hence $a^{p-1} = 1$. Therefore $a^p = a$.

□

A-5 Polynomial Ring

- Reference 1: Polynomial Ring (Wikipedia) [?]
- Reference 2: Polynomial Rings (LibreTexts) [?]

A-5.1 Overview

A polynomial ring is a set of polynomials where polynomial computations over the $(+, \cdot)$ operators (e.g., $f_1 + f_2$, $f_1 \cdot (f_2 - f_3)$, $f_1 + f_2 + f_4$) are closed, associative, commutative, and distributive.

A polynomial ring $\mathbb{Z}_q[x]/(x^n + 1)$ is the set of all polynomials f_i that have the following properties:

⟨Summary ??⟩ Ring

For a polynomial $f \in \mathbb{Z}_q[x]/(x^n + 1)$ where $f = c_0 + c_1x^1 + \ \cdots \ + c_{n-1}x^{n-1}$:

- **Coefficient Ring:** each coefficient $c_j \in \mathbb{Z}_q$.
- **Degree Ring:** Any $f' \in \mathbb{Z}_q[x]$ can be written

  $$f' = (x^n + 1)f_q + f_r, \qquad \deg f_r < n,$$

  so in the quotient ring $\mathbb{Z}_q[x]/(x^n+1)$ we have $f' \equiv f_r \pmod{x^n+1}$. f_q is called a quotient polynomial and f_r is called a remainder polynomial resulting from the polynomial division of f' divided by $x^n + 1$.
- **Polynomial Congruence:** If two polynomials are congruent, they belong to the same equivalence class, in which case they are interchangeable in the polynomial operations $(+, \cdot)$ in the polynomial ring. For example, if:

  $$f' \equiv f_{r1} \in \mathbb{Z}_q[x]/(x^n + 1)$$
  $$f'' \equiv f_{r2} \in \mathbb{Z}_q[x]/(x^n + 1)$$
  $$f_{r1} + f_{r2} \equiv f_{r3} \in \mathbb{Z}_q[x]/(x^n + 1)$$

  Then the polynomial operation result of $f' + f''$ is in the same equivalence class as:
  $f' + f'' \equiv f_{r1} + f_{r2} \equiv f_{r3} \in \mathbb{Z}_q[x]/(x^n + 1)$

To make the notation simple, we denote the polynomial ring $\mathbb{Z}_q[x]/(x^n + 1)$ as $\mathcal{R}_{\langle n,q \rangle}$

Recall that in $\mathbb{Z}_p$, any b writes $b = mp + r$ with $0 \leq r < p$, hence $b \equiv r \pmod{p}$ (the quotient m disappears). Similarly, in a polynomial ring $\mathcal{R}_{\langle n,q \rangle}$, a high-degree polynomial f_{big} can be divided by the polynomial modulo $x^n + 1$, which yields:

$f_{big} = (x^n + 1) \cdot (f_q) + f_r \equiv f_r \in \mathcal{R}_{\langle n,q \rangle}$

, whereas f_q is a quotient polynomial, and f_r is a remainder polynomial. In this case, f_{big} is congruent to (i.e., it is in the same equivalence class as) f_r. Thus, f_q can be eliminated, and f_r (i.e., the simplified version of f_{big}) can be used interchangeably for polynomial operations $(+, \cdot)$ in the polynomial ring. Polynomial simplification (i.e., reduction) in a polynomial ring is done by substituting $x^n \equiv -1$ into f_{big} because $x^n + 1 \equiv 0$ in the polynomial ring (this is the same as the

case of a number ring modulo p, where we reduce a number by substituting 0 for p). This way, a high-degree polynomial f_{big} can be recursively simplified to a polynomial of degree less than n by recursively substituting $x^n \equiv -1$ into f_{big}.

For a polynomial modulo, we normally choose a cyclotomic polynomial $x^n + 1$ (where n is 2^p for some integer p) as the divisor, as it provides computational efficiency.

A-5.1.1 Example

Given $f \in \mathbb{Z}_7[x]/(x^2+1)$, suppose $f = x^4 + 3x^3 + 11x^2 + 6x + 10$. Then,

$$
\begin{aligned}
f &= (x^2) \cdot (x^2) + 3x \cdot (x^2) + 11x^2 + 6x + 10 \\
&\equiv (-1)(-1) + 3x(-1) + (11 \bmod 7)(-1) + 6x + (10 \bmod 7) \\
&= 3x \in \mathbb{Z}_7[x]/(x^2+1)
\end{aligned}
$$

Thus, $f(x) = x^4 + 3x^3 + 11x^2 + 6x + 10$ is equivalent to ($\equiv$) $3x$ in the polynomial ring $\mathbb{Z}_7[x]/(x^2+1)$.

A-5.1.2 Discussion

Congruency: If two numbers are congruent, they belong to the same *congruence class*. The same is true for two congruent polynomials. If the computation results of two mathematical formulas belong to the same congruency class, then their computations wrap around within the modulus of their congruency. This is a useful property for cryptographic schemes where encryption & decryption computations wrap around their values within a limited set of binary bits. Congruency is useful for simplifying computations. For example, a large number or a complex polynomial can be *normalized* to a simpler number or polynomial by using the congruency rule, which reduces the computational overhead.

Polynomial Evaluation: Note that two numbers that belong to the same congruence class are not necessarily the same number. For example, $5 \equiv 10$ modulo 5, but these two numbers are not the same. Likewise, two congruent polynomials are not the same. While two congruent polynomials in a polynomial ring can be interchangeably used for polynomial operations supported in the ring (i.e., $(+, \cdot)$), such as $f_1 + f_2$ or $f_1 \cdot (f_2 - f_3)$, two congruent polynomials do not necessarily yield the same result when evaluated for a certain variable value $x = a$. For example, in the previous example of **??**, the two polynomials $x^4+3x^3+11x^2+6x+10$ and $3x$ are congruent in the polynomial ring $\mathbb{Z}_7[x]/(x^2+1)$. However, these two polynomials do not give the same evaluation results for $x = 0$: the original polynomial gives 10, whereas the reduced (i.e., simplified) polynomial gives 0. This discrepancy in evaluation occurs because we defined two polynomials M_1 and M_2 to be congruent over $x^n + 1$ (i.e., $M_1 \equiv M_2$) if their remainder is the same after being divided by $x^n + 1$ (i.e., $M_1 = Q \cdot (x^n + 1) + M_2$ for some polynomial Q). Therefore, M_1 and M_2 will be evaluated to the same polynomial M_2 if they are evaluated at the x values such that $x^n = -1$, which makes the $x^n + 1$ term 0. The solutions for $x^n = -1$ are called the n-th roots of unity, which we will learn in **??** and **??**. We summarize as follows:

	Ring	Polynomial Ring
Set Elements	number	polynomial
Ring Notation & Definition	$\mathbb{Z}_p = \{0, 1, \ldots, p-1\}$ The set of all integers between 0 and p	$\mathbb{Z}_p[x]/(x^n+1)$ The set of all polynomials f such that $f = c_0 + c_1x^1 + c_2x^2 \cdots + c_{n-1}x^{n-1}$ where each coefficient $c_i \in \mathbb{Z}_p$ and f's degree is less than n
Supported Operations	$(+, \cdot)$ (Addition, Multiplication)	$(+, \cdot)$ (Addition, Multiplication)
$(+)$ Operation	We know how to add numbers	$f_a = a_0 + a_1x^1 + a_2x^2 \cdots + a_{d_a-1}x^{d_a-1}$ $f_b = b_0 + b_1x^1 + b_2x^2 \cdots + b_{d_b-1}x^{d_b-1}$ Then, $f_a + f_b$ is computed as: $f_c = \sum_{i=0}^{\max(d_a,d_b)} (a_i + b_i)x^i$
$(\cdot)$ Operation	We know how to multiply numbers	$f_a = a_0 + a_1x^1 + a_2x^2 \cdots + a_{d_a-1}x^{d_a-1}$ $f_b = b_0 + b_1x^1 + b_2x^2 \cdots + b_{d_b-1}x^{d_b-1}$ Then, $f_a \cdot f_b$ is computed as: $f_c = \sum_{i=0}^{d_a+d_b} \sum_{j=0}^{i} a_j b_{i-j} x^i$
Commutative Associative Distributive Closed	For $a, b \in \mathbb{Z}_p$ $a + b = b + a$ $(a + b) + c = a + (b + c)$ $a \cdot (b + c) = a \cdot b + a \cdot c$ $a + b \equiv c \in \mathbb{Z}_p$, $a \cdot b \equiv d \in \mathbb{Z}_p$	For $f_a, f_b \in \mathbb{Z}_p[x]/(x^n+1)$, $f_a + f_b = f_b + f_a$ $(f_a + f_b) + f_c = f_a + (f_b + f_c)$ $f_a \cdot (f_b + f_c) = f_a \cdot f_b + f_a \cdot f_c$ $f_a + f_b \equiv f_c \in \mathcal{R}_{\langle n,q \rangle}$, $f_a \cdot f_b \equiv f_d \in \mathcal{R}_{\langle n,q \rangle}$
Congruency ($\equiv$)	Two numbers $a \equiv b$ in modulo p if: $(a \bmod p) = (b \bmod p)$	Two polynomials $f_a \equiv f_b$ in $\mathbb{Z}_p[x]/(x^n+1)$ if: $f'_a = f_a \bmod (x^n+1) = \sum_{i=0}^{d_a} a_i x^i$ $f'_b = f_b \bmod (x^n+1) = \sum_{i=0}^{d_b} b_i x^i$, $d_a = d_b$ and $a_i \equiv b_i$ in modulo p for all $0 \le i \le d_a$

Table 1: Comparison between a number ring and a polynomial ring.

⟨Summary ??⟩ Polynomial Evaluation over Polynomial Ring

Suppose polynomials M_1 and M_2 are congruent over the polynomial ring $x^n + 1$. This is equivalent to the following relation: $M_1 = Q \cdot (x^n + 1) + M_2$ Q for some polynomial Q. Then, $M_1(X)$ and $M_2(X)$ are guaranteed to be evaluated to the same value if $X = x_i$ is the solution for $x^n + 1$ (i.e., X is the n-th root of unity). That is , $M_1(x_i) = M_2(x_i)$.

Number Ring & Polynomial Ring: These two rings share many common characteristics, which are summarized in **??**.

A-5.2 Coefficient Rotation

Coefficient rotation is a process of shifting the entire coefficients of a polynomial (either to the left or right) in a polynomial ring. In order to rotate the entire coefficients of a polynomial by h positions to the left, we multiply x^{-h} with the polynomial.

For example, suppose we have a polynomial as follows:

$f(x) = c_0 + c_1x^1 + c_2x^2 + \ \cdots \ + c_hx^h + \ \cdots \ + c_{n-1}x^{n-1} \in \mathcal{R}_{\langle n,q \rangle}$

To shift the entire coefficients of f to the left by h positions (i.e., shift f's h-th coefficient to the constant term), we compute $f \cdot x^{-h}$, which is:

⟨Summary ??⟩ Polynomial Rotation

Given the $(n-1)$-degree polynomial:

$f(x) = c_0 + c_1x^1 + c_2x^2 + \ \cdots \ + c_hx^h + \ \cdots \ + c_{n-1}x^{n-1} \in \mathcal{R}_{\langle n,q \rangle}$

The coefficients of $f(x)$ can be rotated to the left by h positions by multiplying to $f(x)$ by x^{-h} as follows:

$f(x) \cdot x^{-h} = c_0 \cdot x^{-h} + c_1x^1 \cdot x^{-h} + c_2x^2 \cdot x^{-h} + \ \cdots \ + c_hx^h \cdot x^{-h} + \ \cdots \ + c_{n-1}x^{n-1} \cdot x^{-h}$
$\equiv c_h + c_{h+1}x + c_{h+2}x^2 + \ \cdots \ + c_{n-1}x^{n-1-h} - c_0x^{n-h} - \ \cdots \ - c_{h-1}x^{n-1} \in \mathcal{R}_{\langle n,q \rangle}$

Note that multiplying the two polynomials f and x^{-h} will yield a congruent polynomial in $\mathcal{R}_{\langle n,q \rangle}$. Therefore, the rotated polynomial, which is the result of $f \cdot x^{-h}$, will also have a congruent polynomial in $\mathcal{R}_{\langle n,q \rangle}$.

Note that the coefficient signs change when they rotate around the boundary of $x^n(= -1)$, as the computation is conducted in the polynomial ring $Z_q[x]/(x^n + 1)$.

A-5.2.1 Example

Suppose we have a polynomial $f \in \mathbb{Z}_8[x]/(x^4 + 1)$ as follows:

We use the centered residue system for $\mathbb{Z}_8$, i.e., $\{-4, -3, -2, -1, 0, 1, 2, 3\}$.

$f = 2 + 3x - 4x^2 - x^3$

The polynomial ring $\mathbb{Z}_8[x]/(x^4 + 1)$ has the following 4 congruence relationships:

$\boldsymbol{x^4 \equiv -1}$
$x^4 \cdot x^{-1} \equiv -1 \cdot x^{-1}$
$\boldsymbol{x^3 \equiv -x^{-1}}$

$x^4 \equiv -1$
$x^4 \cdot x^{-3} \equiv -1 \cdot x^{-3}$
$\boldsymbol{x \equiv -x^{-3}}$

$x^4 \equiv -1$
$x^4 \cdot x^{-2} \equiv -1 \cdot x^{-2}$
$\boldsymbol{x^2 \equiv -x^{-2}}$

Then, based on the coefficient rotation technique in Summary **??**.1, rotating 1 position to the left is equivalent to computing $f \cdot x^{-1}$ as follows:

$f \cdot x^{-1} = 2 \cdot (x^{-1}) + 3x \cdot (x^{-1}) - 4x^2 \cdot (x^{-1}) - x^3 \cdot (x^{-1})$
$\equiv -2x^3 + 3 - 4x^1 - x^2$
$= 3 - 4x^1 - x^2 - 2x^3$

As another example, rotating 3 positions to the left is equivalent to computing $f \cdot x^{-3}$ as follows:

$f \cdot x^{-3} = 2 \cdot (x^{-3}) + 3x \cdot (x^{-3}) - 4x^2 \cdot (x^{-3}) - x^3 \cdot (x^{-3})$
$\equiv -2x - 3x^2 + 4x^3 - 1$
$= -1 - 2x - 3x^2 + 4x^3$
$= -1 - 2x - 3x^2 + (4 \equiv -4 \bmod 8)x^3$
$\equiv -1 - 2x - 3x^2 - 4x^3$

A-6 Decomposition

Decomposition is a mathematical technique used to convert a large-base number into a mathematical formula that expresses the same value in a smaller base. This section will explain number decomposition and polynomial decomposition.

A-6.1 Number Decomposition

We fix a modulus $q \geq 2$ and write $\mathbb{Z}_q = \mathbb{Z}/q\mathbb{Z}$. Let $\gamma \in \mathbb{Z}_q$. Number decomposition expresses γ as a sum of multiple numbers in base β as follows:

$$\gamma = \gamma_1 \frac{q}{\beta^1} + \gamma_2 \frac{q}{\beta^2} + \cdots + \gamma_\ell \frac{q}{\beta^\ell}$$

where $\beta \geq 2$ is a base and $\ell \geq 1$ is the decomposition level. We assume $\beta^\ell \mid q$ and take digits $\gamma_i \in \{0, 1, \ldots, \beta - 1\}$; under these conditions, the decomposition is unique. This is visually shown in **??**. (If $\beta^\ell \nmid q$, see **??**.) Each γ_i represents a base-β digit at position i. When q is a power of two, this corresponds to a shift by $i \cdot \log_2 \beta$ bits.

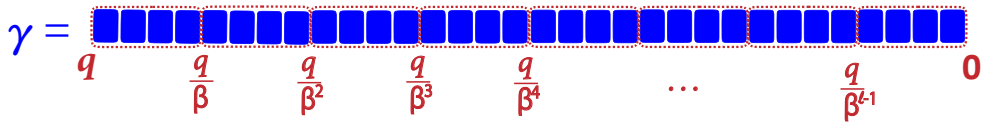


Figure 1: An illustration of number decomposition.

We define the decomposition of the number γ into base β with level ℓ as follows:

$$\mathsf{Decomp}^{\beta,\ell}(\gamma) = (\gamma_1, \gamma_2, \ \cdots, \ \gamma_\ell).$$

Number decomposition is also called radix decomposition, where the base β is referred to as a radix.

A-6.1.1 Example

Suppose we take $\gamma = 13$ in $\mathbb{Z}_{16}$. Suppose we want to decompose 13 with the base $\beta = 2$ and level $\ell = 4$. Then, 13 is decomposed as follows:

$$13 = 1 \cdot \frac{16}{2^1} + 1 \cdot \frac{16}{2^2} + 0 \cdot \frac{16}{2^3} + 1 \cdot \frac{16}{2^4}$$

$$\mathsf{Decomp}^{2,4}(13) = (1, 1, 0, 1)$$

A-6.2 Polynomial Decomposition

This time, suppose we have a polynomial f in the polynomial ring $\mathbb{Z}_q[x]/(x^n + 1)$. Therefore, each coefficient c_i of f is an integer modulo q. Polynomial decomposition expresses f as a sum of multiple polynomials in base β and level ℓ as follows:

⟨Summary ??⟩ Polynomial Decomposition

Given $f \in \mathbb{Z}_q[x]/(x^n+1)$, fix $\beta \geq 2$ and $\ell \geq 1$ with $\beta^\ell \mid q$. We write

$$f = \sum_{i=1}^{\ell} f_i \frac{q}{\beta^i}, \qquad f_i \in \mathbb{Z}_q[x]/(x^n+1)$$

where each f_i is obtained by decomposing every coefficient of f in base β. If $f = \sum_j c_j x^j$ with $c_j \in \mathbb{Z}_q$, then $c_j = \sum_{i=1}^{\ell} c_{j,i} \frac{q}{\beta^i}$ with $c_{j,i} \in \{0, \ldots, \beta-1\}$, and $f_i = \sum_j c_{j,i} x^j$. We denote the decomposition of the polynomial f into the base β with the level ℓ as follows:

$$\mathsf{Decomp}^{\beta,\ell}(f) = (f_1, f_2, \ \cdots, \ f_\ell)$$

A-6.2.1 Example

Suppose we have the following polynomial in the polynomial ring $\mathbb{Z}_{16}[x]/(x^4+1)$:

$$f = 6x^3 + 3x^2 + 14x + 7 \in \mathbb{Z}_{16}[x]/(x^4+1)$$

Suppose we want to decompose the above polynomial with base $\beta = 4$ and level $\ell = 2$. Then, each degree's coefficient of the polynomial f is decomposed as follows:

$$\boldsymbol{x^3}\text{: } 6 = 1 \cdot \frac{16}{4^1} + 2 \cdot \frac{16}{4^2}$$
$$\boldsymbol{x^2}\text{: } 3 = 0 \cdot \frac{16}{4^1} + 3 \cdot \frac{16}{4^2}$$
$$\boldsymbol{x^1}\text{: } 14 = 3 \cdot \frac{16}{4^1} + 2 \cdot \frac{16}{4^2}$$
$$\boldsymbol{x^0}\text{: } 7 = 1 \cdot \frac{16}{4^1} + 3 \cdot \frac{16}{4^2}$$

The decomposed polynomial is as follows:

$$f = 6x^3 + 3x^2 + 14x + 7 = (1x^3 + 0x^2 + 3x + 1) \cdot \frac{16}{4^1} + (2x^3 + 3x^2 + 2x + 3) \cdot \frac{16}{4^2}$$

$$\mathsf{Decomp}^{4,2}(6x^3 + 3x^2 + 14x + 7) = (1x^3 + 0x^2 + 3x + 1, 2x^3 + 3x^2 + 2x + 3)$$

A-6.2.2 Discussion

Note that after decomposition, the original coefficients of the polynomial have been reduced to smaller numbers. This characteristic is importantly used in the multiplication of polynomials in FHE ciphertexts to reduce the growth rate of the noise. Normally, the polynomial coefficients of ciphertexts are large because they are uniformly random numbers. Reducing such large constants is important for reducing the noise growth during homomorphic multiplication. We will discuss this in more detail in **??**.

A-6.3 Approximate Decomposition

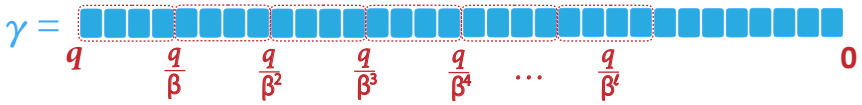


Figure 2: An illustration of approximate decomposition

If no level ℓ satisfies $\beta^\ell \mid q$, then some lower bits of q must be discarded during decomposition, as shown in **??**. Such lower bits can be rounded to the nearest multiple of $\frac{q}{\beta^\ell}$ during decomposition. In such a case, the decomposition is an approximate decomposition. Formally, when $\beta^\ell \nmid q$ we can write

$$\gamma = \sum_{i=1}^{\ell} \gamma_i \frac{q}{\beta^i} + \varepsilon, \qquad \gamma_i \in \{0, \ldots, \beta - 1\}, \quad |\varepsilon| \leq \frac{q}{2\beta^\ell}$$

(using nearest-integer rounding and identifying γ with its integer representative)The polynomial case is analogous, coefficient-wise.

A-6.4 Gadget Decomposition

Gadget decomposition is a generalized form of number decomposition **(??)**. In number decomposition, a number γ is decomposed as follows:

$\gamma = \gamma_1 \frac{q}{\beta^1} + \gamma_2 \frac{q}{\beta^2} + \cdots + \gamma_\ell \frac{q}{\beta^\ell}$

In gadget decomposition, we decompose γ as follows:

$\gamma = \gamma_1 g_1 + \gamma_2 g_2 + \cdots + \gamma_\ell g_\ell$

We denote $\vec{g} = (g_1, g_2, \cdots, g_\ell)$ as a gadget vector, and $\mathsf{Decomp}^{\vec{g}}(\gamma) = (\gamma_1, \gamma_2, \ \cdots, \ \gamma_\ell)$

Then, $\gamma = \langle \mathsf{Decomp}^{\vec{g}}(\gamma), \vec{g} \rangle$

In the case of number decomposition **(??)**, its gadget vector is $\vec{g} = \left(\frac{q}{\beta}, \frac{q}{\beta^2}, \cdots, \frac{q}{\beta^\ell} \right)$.

A-7 Roots of Unity

- Reference: Fields and Cyclotomic Polynomials [**?**]

A-7.1 Definitions

⟨Definition ??⟩ Definitions for Roots of Unity

- ***n*-th root of Unity:** A complex number ζ that satisfies the equation $\zeta^n = 1$
- **Primitive *n*-th Root of Unity:** Any n-th root of unity ζ such that $\mathsf{ord}_{\mathbb{C}}(\zeta) = n$. We denote $P(n)$ as a set of primitive n-th root of unity.

A primitive n-th root of unity is considered a *generator* of all n n-th roots of unity.

A-7.2 Theorems

⟨Theorem ??.1⟩ Formula for *n*-th Root of Unity

Given $\zeta^n = 1$, there exist exactly n different n-th roots of unity:

$$\zeta = e^{2k\pi i/n} = \cos\left(\frac{2k\pi}{n}\right) + i \cdot \sin\left(\frac{2k\pi}{n}\right),$$

for n different k values, where $k = \{0, 1, \cdots, n-1\}$.

Proof.

1. Suppose $\zeta = e^{2k\pi i/n}$. Then, $\zeta^n = (e^{2k\pi i/n})^n = e^{2k\pi i}$, and since $\zeta^n = 1$, we need to find the k values such that $e^{2k\pi i} = 1$
2. Euler's formula states that $e^{i \cdot x} = \cos(x) + i \cdot \sin(x)$. Therefore, if $x = 2k\pi$, then $e^{2k\pi i} = \cos(2k\pi) + i \cdot \sin(2k\pi)$. This formula becomes 1 if $k = 0, 1, 2, ...$ Thus, $e^{2k\pi i} = 1$ for any integer $k \geq 0$.
3. If $\zeta = e^{2k\pi i/n} = \cos(\frac{2k\pi}{n}) + i \cdot \sin(\frac{2k\pi}{n})$, then the first n roots for $k = 0, 1, ... n-1$ are all distinct values, because they lie on the circle in the complex plane (where x-axis is a real value and y-axis is a complex value coefficient) at each angle $2k\pi/n$ for $k = \{0, 1, \cdots, n-1\}$.
4. Note $\zeta^n = 1$ is an n-th polynomial, so it can have at most n roots. Thus, we can consider the first n roots $e^{2k\pi i/n}$ for $k = \{0, 1, \cdots, n-1\}$ as the n distinct roots and ignore the rest of roots (i.e., $k \geq n$), considering them to be repetitions of the first n roots on a circle in the complex plane (see **??**).

□

⟨Theorem ??.2⟩ Order of the Root of Unity

Given $\zeta \in \mathbb{C}$ (the complex number domain) and $\zeta^n = 1$ where $n \geq 1$, ζ is an n-th root of unity if and only if $\mathsf{ord}_{\mathbb{C}}(\zeta) \mid n$.

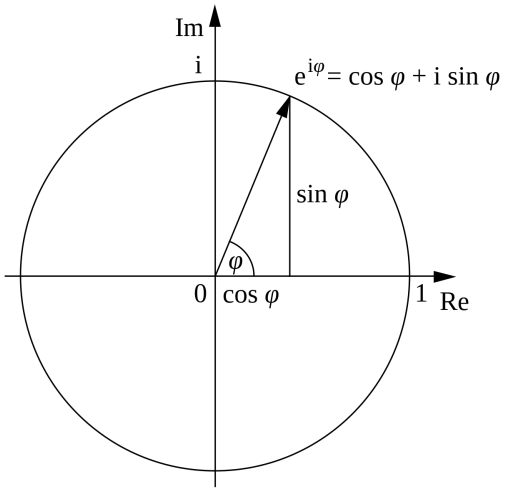


Figure 3: The figure illustrates a circle of Euler's formula in the complex plane (Source)

Proof. We use Theorem **??**.1:

1. *Forward Proof:* Since $\mathsf{ord}_{\mathbb{C}}(\zeta) = k$ is the smallest integer such that $\zeta^k = 1$, for any n that satisfies $\zeta^n = 1$, n must be a multiple of k. This means that $k \mid n$.
2. *Backward Proof:* If $k \mid n$, then n is a multiple of k, which means that $\zeta^n = 1$.

□

⟨Theorem ??.3⟩ Set of All n-th Roots of Unity

The set of all n-th roots of unity is the union $\bigcup_{d|n} P(d)$ (i.e., the union of all primitive d-th roots of unity where $d \mid n$).

Proof.

1. Let $\omega = e^{2\pi i/n}$. Given $\zeta^n = 1$, for each n-th root of unity ζ is, $\zeta = \omega^{k_i}$ for $k_i = \{0, 1, \cdots, n-1\}$. Note that according to Theorem **??**.1, $(\omega^{k_i})^n = 1$ if and only if $\mathsf{ord}_{\mathbb{C}}(\omega^{k_i}) \mid n$.
2. Let $\mathsf{ord}_{\mathbb{C}}(\omega^{k_i}) = d_i$. Then, $(\omega^{k_i})^{d_i} = 1$. Combining these two facts, each n-th root of unity ω^{k_i} is also a primary d_i-th root of unity (i.e., a solution for $\zeta^{d_i} = 1$), that is, $\omega^{k_i} \in P(d_i)$.
3. Remember that for each $\mathsf{ord}_{\mathbb{F}}(\omega^{k_i}) = d_i$, $d_i \mid n$. For every d_i that divides n, all the (primary) d_i-th roots of unity are also the n-th root of unity. This is because the (primary) d_i-th root of unity that satisfies $\zeta^{d_i} = 1$ also satisfies $\zeta^n = 1$ (as n is a multiple of d_i).
4. Step 2 concludes that each n-th root of unity is a primitive d_i-th root of unity for some d_i that divides n. Step 3 concludes that each d_i-th root of unity, where d_i divides n, is also the n-th root of unity. Combining these two conclusions, the set of all primitive n-th root of unity is equivalent to the union of all primary d_i-th roots of unity where d_i divides n (i.e., $\bigcup_{d|n} P(d)$).

□

⟨Theorem ??.4⟩ Condition for Primitive n-th Roots of Unity

Given an n-th root of unity $\zeta = \omega^k$ for $k = \{0, 1, \cdots, n-1\}$ where $\omega = e^{2\pi i/n}$, ζ is a primitive n-th root of unity if and only if $\gcd(n, k) = 1$ (i.e., k is co-prime to n).

Proof.

1. Note that $\zeta^n = 1$ and $\zeta = \omega$ for $k = 1$. Thus, $\mathsf{ord}_{\mathbb{C}}(\omega) = n$.
2. Theorem **??**.2 states that if $ord_{\mathbb{F}}(a) = k$, then for any $n \geq 1$, $ord_{\mathbb{F}}(a^n) = \dfrac{k}{\gcd(k,n)}$. Similarly, if $\mathsf{ord}_{\mathbb{C}}(\omega) = n$, then for any $k \geq 1$, $\mathsf{ord}_{\mathbb{C}}(\omega^k) = \dfrac{n}{\gcd(k,n)}$.
3. Step 2 implies that $\mathsf{ord}_{\mathbb{C}}(\omega^k) = n$ (i.e., ω^k is a primitive n-th root of unity) if and only if $\gcd(k,n) = 1$.

□

⟨Theorem ??.5⟩ The number of Primitive n-th Roots of Unity

The number of primitive n-th roots of unity is $\phi(n)$ (i.e., the number of elements in $\{1, \cdots, n-1\}$ that are coprime to n).

Proof.

1. Given $\zeta^n = 1$, the roots of unity are $\zeta = \omega^k$ where $\omega = e^{2\pi i/n}$ and $k = \{0, 1, \cdots, n-1\}$
2. By definition, ω^k is a primitive n-th root of unity if and only if $\mathsf{ord}_{\mathbb{C}}(\omega^k) = n$.
3. ω is a primitive n-th root of unity because $\mathsf{ord}_{\mathbb{C}}(\omega) = n$.
4. According to Theorem **??**.2, if $\mathsf{ord}_{\mathbb{C}}(\omega) = n$, then $\mathsf{ord}_{\mathbb{C}}(\omega^k) = \dfrac{n}{\gcd(k,n)}$. Therefore, in order for $\mathsf{ord}_{\mathbb{C}}(\omega^k) = n$, $\gcd(k,n)$ has to be 1. In other words, k and n have to be co-prime. The total number of such co-primes between n and $k = \{1, 2, \cdots, n-1\}$ (excluding 0 because $\gcd(0,n) = n$ and also $\mathsf{ord}_{\mathbb{C}}(\omega^0) = \mathsf{ord}_{\mathbb{C}}(1) = 1 \neq n$) is $\phi(n)$, which corresponds to the total number of the primitive n-th root of unity.

□

A-8 Cyclotomic Polynomial

- Reference: Fields and Cyclotomic Polynomials [**?**]

A-8.1 Definitions

⟨Definition ??⟩ Cyclotomic Polynomial

The n-th Cyclotomic Polynomial: is a polynomial whose roots are the primitive n-th root of unity, that is:

$$\Phi_n(x) = \prod_{\zeta \in P(n)} (x - \zeta) = \prod_{\substack{0 \le k \le n-1, \\ \gcd(k,n)=1}} (x - \omega^k) \quad \text{..... , where } \omega = e^{2\pi i/n}$$

Remember the Euler's formula: $e^{2k\pi i/n} = \cos\left(\frac{2k\pi}{n}\right) + i \cdot \sin\left(\frac{2k\pi}{n}\right)$

A few pre-computed cyclotomic polynomials are as follows:

$\Phi_1(x) = x - 1$
$\Phi_2(x) = x + 1$
$\Phi_3(x) = x^2 + x + 1$
$\Phi_4(x) = x^2 + 1$
$\Phi_5(x) = x^4 + x^3 + x^2 + x + 1$
$\Phi_6(x) = x^2 - x + 1$
$\Phi_7(x) = x^6 + x^5 + x^4 + x^3 + x^2 + 1$
$\Phi_8(x) = x^4 + 1$
$\Phi_9(x) = x^6 + x^3 + 1$
$\Phi_{10}(x) = x^4 - x^3 + x^2 - x + 1$

As one example,

$$\Phi_4(x) = \prod_{\substack{0 \le k \le 3, \\ \gcd(k,4)=1}} (x - \omega^k) = (x - \omega^1)(x - \omega^3) = (x - e^{2\pi i/4})(x - e^{2\cdot 3\pi i/4}) = (x - e^{\pi i/2})(x - e^{\cdot 3\pi i/2})$$

$$= \left(x - \left(\cos\left(\frac{\pi}{2}\right) + i \cdot \sin\left(\frac{\pi}{2}\right)\right)\right) \cdot \left(x - \left(\cos\left(\frac{3\pi}{2}\right) + i \cdot \sin\left(\frac{3\pi}{2}\right)\right)\right)$$
$$= (x - i)(x + i) = x^2 + 1$$

A-8.2 Theorems

⟨Theorem ??.1⟩ Roots of the M-th Cyclotomic Polynomial

Suppose that M is a power of 2 and the M-th cyclotomic polynomial $\Phi_M(x) = x^n + 1$ (where $M = 2n$). Then, the roots of the M-th cyclotomic polynomial are $\omega, \omega^3, \omega^5, \cdots, \omega^{2n-1}$, where $\omega = e^{i\pi/n}$

Proof. According to Definition **??** in **??**, the roots of $\Phi_M(x)$ are $e^{2k\pi i/M} = e^{2k\pi i/(2n)} = e^{k\pi i/n}$ where $0 \le k < M = 2n$ and $\gcd(k, M = 2n) = 1$, thus $k = \{1, 3, 5, \cdots, 2n-1\}$. If we let $\omega = e^{i\pi/n}$, then the roots of $\Phi_M(x)$ are $\omega, \omega^3, \omega^5, \cdots, \omega^{2n-1}$.

□

⟨Theorem ??.2⟩ Polynomial Decomposition into Cyclotomic Polynomials

For any positive integer n,

$$x^n - 1 = \prod_{d \mid n} \Phi_d(x)$$

Proof.

1. The roots of $x^n - 1$ are all the n-th roots of unity. Thus, $x^n - 1 = (x - \omega^0)(x - \omega^1)...(x - \omega^{n-1})$, where $\zeta = \omega^k$.
2. Theorem **??**.3 states that each n-th root of unity (ω^k) is a primitive d-th root of unity for some d that divides n. In other words, each n-th root of unity belongs to some $P(d)$ where $d \mid n$. Meanwhile, by definition, $\Phi_d(x) = \Pi_{\zeta \in P(d)}(x - \zeta)$. Therefore, $x^n - 1$ is a multiplication of all $\Phi_d(x)$ such that $d \mid n$.

□

⟨Theorem ??.3⟩ Integer Coefficients of Cyclotomic Polynomials

A cyclotomic polynomial has only integer coefficients.

Proof.

1. We prove by induction. When $n = 1$, $\Phi_1(X) = x - 1$, where each coefficient is an integer.
2. Let $x^n - 1 = f(x) \cdot g(x) = (\Sigma_{i=0}^{p} a_i x^i)(\Sigma_{j=0}^{q} b_j x^i)$. As an induction hypothesis 1, we will prove that if $f(x)$ has only integer coefficients, then $g(x)$ will also have only integer coefficients. Given our target equation is $x^n - 1$, we know that $a_p x_p \cdot b_q x_q = x^n$, and thus $a_p b_q = 1$, which means $a_p = \pm 1$ (as we hypothesized that $f(x)$ has only integer coefficients). We also know that $a_0 b_0 = -1$. All the other coefficients should be 0. Thus, for any $r < q$, the coefficients are either: (i) $a_p b_r + a_{p-1} b_{r+1} + ... + a_{p-q+r} b_q = 0$; or (ii) $a_p b_r + a_{p-1} b_{r+1} + ... + a_0 b_{r+p} = 0$. Both case (i) and (ii) represent $f(x) \cdot g(x)$'s computed coefficient of some x^i where $0 < i < n$. Now, we propose another induction hypothesis 2, which is that $b_q, ... b_{r+1}$ are all integers.
3. In the case of (i), $a_p b_r = -(a_{p-1} b_{r+1} + ... + a_{p-q+r} b_q)$, and dividing both sides by a_p (which is either 1 or -1), $b_r = \pm(a_{p-1} b_{r+1} + ... + a_{p-q+r} b_q)$, as every a_i is an integer based on our hypothesis. By induction hypothesis 1 and 2, b_r is an integer. The same is true in the case of (ii).
4. We set b_q (an integer coefficient) as the starting point for induction hypothesis 2. Then, according to induction proof 2, all of b_j for $0 \leq j \leq q$ are integers.
5. Now, we set $\Phi_1(X)$ (an integer coefficient polynomial) as the starting point for induction hypothesis 1. Let $x^n - 1 = \Phi_{d_1}(X)\Phi_{d_2}(X)...\Phi_{d_k}(X)\Phi_n(X)$, where each $d_i \mid n$ (Theorem **??**.1). We know that $\Phi_{d_1}(X)\Phi_{d_2}(X) \cdots \Phi_{d_k}(X)$ forms an integer coefficient polynomial. We treat $\Phi_{d_1}(X)\Phi_{d_2}(X) \cdots \Phi_{d_k}(X)$ as $f(x)$, and $\Phi_n(X)$ as $g(x)$. Then, according to step 4's induction proof, $\Phi_n(X)$ is an integer coefficient polynomial (also note that $\Phi_n(X)$ is monoic, whose the highest degree's coefficient is 1).
6. As we marginally increase n to $n + 1$ to compute $x^{n+1} - 1 = \Phi_{d'_1}(X)\Phi_{d'_2}(X)...\Phi_{d'_k}(X)\Phi_{n+1}(X)$ (where each $d'_i \mid (n + 1)$), we know that $\Phi_{d'_1}(X)\Phi_{d'_2}(X) \cdots \Phi_{d'_k}(X)$ is a monoic polynomial, as proved by the previous induction step. Thus, $\Phi_{n+1}(X)$ is also monoic.

□

⟨Theorem ??.4⟩ Formula for $\Phi_{nk}(x)$

If $k \mid n$, then $\Phi_{nk}(x) = \Phi_n(x^k)$.

Proof.

1. Theorem **??**.3 states that given $k \mid n$, $\text{ord}_{\mathbb{F}}(a) = kn$ if and only if $\text{ord}_{\mathbb{F}}(a^k) = n$. This means that for $\zeta \in \mathbb{C}$, $\text{ord}_{\mathbb{C}}(\zeta) = nk$ if and only if $\text{ord}_{\mathbb{C}}(\zeta^k) = n$. In other words, ζ is a primitive nk-th root of unity if and only if ζ^k is the primitive n-th root of unity. This implies that ζ is a root of $\Phi_{nk}(x)$ if and only if ζ^k is a root of $\Phi_n(x)$.
2. Let $\Phi_{nk}(x) = (x - \zeta_1)(x - \zeta_2)...(x - \zeta_p)$, where $P(n)$ has p primitive nk-th roots of unity.
3. $\Phi_n(x) = (x - \zeta_1^k)(x - \zeta_2^k)...(x - \zeta_p^k)$. Note that $P(n)$ should also have p primitive n-th roots of unity, because elements of $P(nk)$ are isomorphic to the elements of $P(n)$ (as they preserved if and only if relationships in step 2). Now, it's also true that $\Phi_n(y) = (y - \zeta_1^k)(y - \zeta_2^k)...(y - \zeta_p^k)$, where $y = x^k$. In this case, $x = \{\zeta_1, \zeta_2, ...\zeta_p\}$.
4. $\Phi_{nk}(x)$ and $\Phi_n(y) = \Phi_n(x^k)$ have the same roots with the same coefficients. Therefore, $\Phi_{nk}(x) = \Phi_n(y) = \Phi_n(x^k)$.

□

A-9 Roots of Unity and Cyclotomic Polynomial over Rings

In **??** and **??**, we learned about the definition and properties of the μ-th roots of unity and the μ-th cyclotomic polynomial over complex numbers (i.e., $X \in \mathbb{C}$) as follows:

- **The μ-th roots of unity** are the solutions for $X^\mu = 1$ over $X \in \mathbb{C}$ (complex numbers). The formula for the μ-th root of unity is $X = e^{2\pi ik/\mu}$ for all integers k such that $0 \leq k \leq \mu - 1$.
- **The primitive μ-th roots of unity (denoted as ω)** are those μ-th roots of unity whose order (**??**) is μ (i.e., $\omega^\mu = 1$ and $\omega^{\frac{\mu}{2}} \neq 1$).
- Given any primitive μ-th roots of unity ω, one can generate all primitive μ-th roots of unity by computing $\omega^{k'}$ such that k' is an integer $0 < k' < \mu$ and $\mathsf{gcd}(k', \mu) = 1$ (Theorem **??**.4 in **??**).
- **The μ-th cyclotomic polynomial** is defined as a polynomial whose roots are the primitive μ-th roots of unity. That is,

$$\Phi_\mu(x) = \prod_{\omega \in P(\mu)} (x - \omega) = \prod_{\substack{0 \leq k \leq \mu - 1, \\ \gcd(k,\mu)=1}} (x - \omega^k)$$

In this section, we will explain the μ-th cyclotomic polynomial over $X \in \mathbb{Z}_p$ (ring), which is structured as follows:

⟨Definition ??⟩ Roots of Unity and Cyclotomic Polynomial over Rings $\mathbb{Z}_p$

- **The μ-th roots of unity (denoted as ω)** are the solutions for $X^\mu \equiv 1 \bmod p$. Note that these solutions are not $X = \omega^{2\pi ik/\mu}$ (the formula for the solutions over $X \in \mathbb{C}$).
- **The primitive μ-th roots of unity** are defined as those μ-th roots of unity whose order (**??**) is μ (i.e., $\omega^\mu \equiv 1 \bmod p$, and $\omega^{\lceil \frac{\mu}{2} \rfloor} \not\equiv 1 \bmod p$).
- Given any primitive μ-th roots of unity ω, it can generate all primitive μ-th roots of unity by computing $\omega^{k'}$ such that k' is an integer $0 < k' < \mu$ and $\mathsf{gcd}(k', \mu) = 1$.
- **The μ-th cyclotomic polynomial** is defined as a polynomial whose roots are the primitive μ-th roots of unity. That is,

$$\Phi_\mu(x) = \prod_{\omega \in P(\mu)} (x - \omega) = \prod_{\substack{0 \leq k \leq \mu - 1, \\ \gcd(k,\mu)=1}} (x - \omega^k)$$

Note that in the μ-th cyclotomic polynomial, in both cases of over $X \in \mathbb{C}$ and over $X \in \mathbb{Z}_p$, each of their roots ω (i.e., the primitive μ-th root of unity) has the order μ (i.e., $\omega^\mu = 1$ over $X \in \mathbb{C}$, and $\omega^\mu \equiv 1 \bmod p$ over $X \in \mathbb{Z}_p$). Also note that each root ω can generate all roots of the μ-th cyclotomic polynomial by computing $\omega^{k'}$ such that $\mathsf{gcd}(k', \mu) = 1$.

?? compares the properties of the roots of unity and the μ-th cyclotomic polynomial over $X \in \mathbb{C}$ (the complex numbers) and over $X \in \mathbb{Z}_p$ (the ring).

	Polynomial over $X \in \mathbb{C}$ (Complex Number)	**Polynomial over $X \in \mathbb{Z}_p$ (Ring)**
Definition of the μ-th Root of Unity	All $X \in \mathbb{C}$ such that $X^\mu = 1$, (which are computed as $X = e^{2\pi ik/\mu}$ for integer k where $0 \leq k \leq \mu - 1$)	All $X \in \mathbb{Z}_p$ such that $X^\mu \equiv 1 \bmod p$
Definition of the Primitive μ-th Root of Unity	Those μ-th roots of unity ω such that $\omega^\mu = 1$, and $\omega^{\frac{\mu}{2}} \neq 1$	Those μ-th roots of unity ω such that $\omega^\mu \equiv 1 \bmod p$, and $\omega^{\frac{\mu}{2}} \not\equiv 1 \bmod p$
Definition of the μ-th Cyclotomic Polynomial	The polynomial whose roots are the μ-th primitive roots of unity as follows: $\Phi_\mu(x) = \prod_{\omega \in P(\mu)}(x - \omega)$ (see Definition **??** in **??**)	
Finding Primitive μ-th Roots of Unity	For $\omega = e^{2\pi i/\mu}$, compute all ω^k such that $0 < k < \mu$ and $\mathsf{gcd}(k, \mu) = 1$ (Theorem **??**.4 in **??**)	Find one satisfactory ω that is a root of the μ-th cyclotomic polynomial, and compute all $\omega^k \bmod p$ such that $0 < k < \mu$ and $\mathsf{gcd}(k, \mu) = 1$

Table 2: The roots of unity and cyclotomic polynomials over $X \in \mathbb{C}$ v.s. over $X \in \mathbb{Z}_p$

A-10 Vector and Matrix

A-10.1 Vector Arithmetic

This section explains the basic arithmetic of vector and matrix computations, as well as advanced concepts such as vector/plane projection and the basis of planes (or spaces).

⟨Definition ??⟩ Vector Arithmetic

- **Addition:** Given two $n \times 1$ vectors (i.e., n-dimensional vectors) composed of n numbers each:
  $\vec{a} = (a_0, a_1, \cdots, a_{n-1})$, $\vec{b} = (b_0, b_1, \cdots, b_{n-1})$

  Vector addition is defined as:
  $\vec{a} + \vec{b} = (a_0 + b_0,\ a_1 + b_1,\ \cdots,\ a_{n-1} + b_{n-1})$

- **Dot Product:** Given two n-dimensional vectors:
  $\vec{a} = (a_0, a_1, \cdots, a_{n-1})$, $\vec{b} = (b_0, b_1, \cdots, b_{n-1})$

  Dot product is defined as:
  $\langle \vec{a}, \vec{b} \rangle = \vec{a} \cdot \vec{b} = a_0 b_0 + a_1 b_1 + \cdots, + a_{n-1} b_{n-1}$.
  More formally, $\vec{a} \cdot \vec{b} = |a||b| \cos\theta$, where θ is the angle between the two vectors. As $\vec{a}$ and $\vec{b}$ point at the same direction, $\vec{a} \cdot \vec{b}$ convergest to a maximized value. As $\vec{a}$ and $\vec{b}$ have an orthogonal direction, $\vec{a} \cdot \vec{b}$ converges to 0.

- **Hadamard Product:** Given two n-dimensional vectors:
  $\vec{a} = (a_0, a_1, \cdots, a_{n-1})$, $\vec{b} = (b_0, b_1, \cdots, b_{n-1})$

  Hadamard product is defined as:
  $\vec{a} \odot \vec{b} = (a_0 b_0,\ a_1 b_1,\ \cdots,\ a_{n-1} b_{n-1})$

- **Hermitian Product:** Given two n-dimensional complex vectors:
  $\vec{a} = (a_0 + i \cdot a'_0,\ a_1 + i \cdot a'_1,\ \cdots,\ a_{n-1} + i \cdot a'_{n-1})$
  $\vec{b} = (b_0 + i \cdot b'_0,\ b_1 + i \cdot b'_1,\ \cdots,\ b_{n-1} + i \cdot b'_{n-1})$

  Hermitian product is a dot product with the 2nd operand as a conjugate (**??**):
  $\langle \langle \vec{a}, \vec{b} \rangle \rangle = \vec{a} \cdot \overline{\vec{b}}$
  $= (a_0 + i \cdot a'_0,\ a_1 + i \cdot a'_1,\ \cdots,\ a_{n-1} + i \cdot a'_{n-1}) \cdot (b_0 - i \cdot b'_0,\ b_1 - i \cdot b'_1,\ \cdots,\ b_{n-1} - i \cdot b'_{n-1})$

A-10.2 Various Types of Matrix

⟨Definition ??⟩ Matrices

- An $n \times n$ **identity matrix** and a **reverse identity matrix** are defined as:

$$I_n = \begin{bmatrix} 1 & 0 & 0 & \cdots & 0 \\ 0 & 1 & 0 & \cdots & 0 \\ 0 & 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & 1 \end{bmatrix}, I_n^R = \begin{bmatrix} 0 & \cdots & 0 & 0 & 1 \\ 0 & \cdots & 0 & 1 & 0 \\ 0 & \cdots & 1 & 0 & 0 \\ \vdots & \iddots & \vdots & \vdots & \vdots \\ 1 & 0 & 0 & \cdots & 0 \end{bmatrix}$$

- The **transpose of a matrix** X is defined as element-wise swapping along the diagonal line, denoted as X^T, which is:

$$X = \begin{bmatrix} a_1 & a_2 & a_3 & \cdots & a_n \\ b_1 & b_2 & b_3 & \cdots & b_n \\ c_1 & c_2 & c_3 & \cdots & c_n \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ m_1 & m_2 & m_3 & \cdots & m_n \end{bmatrix}, X^T = \begin{bmatrix} a_1 & b_1 & c_1 & \cdots & m_1 \\ a_2 & b_2 & c_2 & \cdots & m_2 \\ a_3 & b_3 & c_3 & \cdots & m_3 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ a_n & b_n & c_n & \cdots & m_n \end{bmatrix}$$

- A **Vandermonde matrix** is an $(m+1) \times (n+1)$ matrix defined as:

$$V(x_0, x_1, \cdots, x_m) = \begin{bmatrix} 1 & x_0 & x_0^2 & \cdots & x_0^n \\ 1 & x_1 & x_1^2 & \cdots & x_1^n \\ 1 & x_2 & x_2^2 & \cdots & x_2^n \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_m & x_m^2 & \cdots & x_m^n \end{bmatrix}$$

A-10.3 Matrix Arithmetic

Matrix-to-vector multiplication and matrix-to-matrix multiplication are defined as follows:

⟨Definition ??⟩ Matrix Arithmetic

- **Matrix-to-Vector Multipication:** Given a $m \times n$ matrix A and a n-dimensional vector x:

$$A = \begin{bmatrix} a_{\langle 1,1 \rangle} & a_{\langle 1,2 \rangle} & a_{\langle 1,3 \rangle} & \cdots & a_{\langle 1,n \rangle} \\ a_{\langle 2,1 \rangle} & a_{\langle 2,2 \rangle} & a_{\langle 2,3 \rangle} & \cdots & a_{\langle 2,n \rangle} \\ a_{\langle 3,1 \rangle} & a_{\langle 3,2 \rangle} & a_{\langle 3,3 \rangle} & \cdots & a_{\langle 3,n \rangle} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ a_{\langle m,1 \rangle} & a_{\langle m,2 \rangle} & a_{\langle m,3 \rangle} & \cdots & a_{\langle m,n \rangle} \end{bmatrix} = \begin{bmatrix} \vec{a}_{\langle 1,* \rangle} \\ \vec{a}_{\langle 2,* \rangle} \\ \vec{a}_{\langle 3,* \rangle} \\ \vdots \\ \vec{a}_{\langle m,* \rangle} \end{bmatrix}, \ \vec{x} = (x_1, x_2, \cdots, x_n)$$

  The result of $A \cdot \vec{x}$ is an n-dimensional vector computed as:

$$A \cdot \vec{x} = \left(\vec{a}_{\langle 1,* \rangle} \cdot \vec{x}, \ \vec{a}_{\langle 2,* \rangle} \cdot \vec{x}, \ \cdots, \ \vec{a}_{\langle m,* \rangle} \cdot \vec{x} \right) = \left(\sum_{i=1}^{n} a_{1,i} \cdot x_i, \ \sum_{i=1}^{n} a_{2,i} \cdot x_i, \ \cdots, \ \sum_{i=1}^{n} a_{m,i} \cdot x_i \right)$$

- **Matrix-to-Matrix Multiplication:** Given a $m \times n$ matrix A and a $n \times k$ matrix B:

$$A = \begin{bmatrix} a_{\langle 1,1\rangle} & a_{\langle 1,2\rangle} & a_{\langle 1,3\rangle} & \cdots & a_{\langle 1,n\rangle} \\ a_{\langle 2,1\rangle} & a_{\langle 2,2\rangle} & a_{\langle 2,3\rangle} & \cdots & a_{\langle 2,n\rangle} \\ a_{\langle 3,1\rangle} & a_{\langle 3,2\rangle} & a_{\langle 3,3\rangle} & \cdots & a_{\langle 3,n\rangle} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ a_{\langle m,1\rangle} & a_{\langle m,2\rangle} & a_{\langle m,3\rangle} & \cdots & a_{\langle m,n\rangle} \end{bmatrix}, B = \begin{bmatrix} b_{\langle 1,1\rangle} & b_{\langle 1,2\rangle} & b_{\langle 1,3\rangle} & \cdots & b_{\langle 1,k\rangle} \\ b_{\langle 2,1\rangle} & b_{\langle 2,2\rangle} & b_{\langle 2,3\rangle} & \cdots & b_{\langle 2,k\rangle} \\ b_{\langle 3,1\rangle} & b_{\langle 3,2\rangle} & b_{\langle 3,3\rangle} & \cdots & b_{\langle 3,k\rangle} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ b_{\langle n,1\rangle} & b_{\langle n,2\rangle} & b_{\langle n,3\rangle} & \cdots & b_{\langle n,k\rangle} \end{bmatrix}$$

The result of $A \cdot B$ is a $m \times k$ matrix computed as:

$$A \cdot B = \begin{bmatrix} \sum_{i=1}^{n} a_{\langle 1,i\rangle} b_{\langle i,1\rangle} & \sum_{i=1}^{n} a_{\langle 1,i\rangle} b_{\langle i,2\rangle} & \sum_{i=1}^{n} a_{\langle 1,i\rangle} b_{\langle i,3\rangle} & \cdots & \sum_{i=1}^{n} a_{\langle 1,i\rangle} b_{\langle i,k\rangle} \\ \sum_{i=1}^{n} a_{\langle 2,i\rangle} b_{\langle i,1\rangle} & \sum_{i=1}^{n} a_{\langle 2,i\rangle} b_{\langle i,2\rangle} & \sum_{i=1}^{n} a_{\langle 2,i\rangle} b_{\langle i,3\rangle} & \cdots & \sum_{i=1}^{n} a_{\langle 2,i\rangle} b_{\langle i,k\rangle} \\ \sum_{i=1}^{n} a_{\langle 3,i\rangle} b_{\langle i,1\rangle} & \sum_{i=1}^{n} a_{\langle 3,i\rangle} b_{\langle i,2\rangle} & \sum_{i=1}^{n} a_{\langle 3,i\rangle} b_{\langle i,3\rangle} & \cdots & \sum_{i=1}^{n} a_{\langle 3,i\rangle} b_{\langle i,k\rangle} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \sum_{i=1}^{n} a_{\langle m,i\rangle} b_{\langle i,1\rangle} & \sum_{i=1}^{n} a_{\langle m,i\rangle} b_{\langle i,2\rangle} & \sum_{i=1}^{n} a_{\langle m,i\rangle} b_{\langle i,3\rangle} & \cdots & \sum_{i=1}^{n} a_{\langle m,i\rangle} b_{\langle i,k\rangle} \end{bmatrix}$$

Given the above definitions of matrix and vector arithmetic, the following algebraic properties can be derived:

⟨Theorem ??⟩ Matrix Arithmetic Properties

- **Associative:**
  $(AB)C = A(BC)$
  $A(Bx) = (AB)x$

- **Distributive:**
  $A(x + y) = Ax + Ay$
  $A(x \odot y) = Ax \odot Ay$
  $A\langle x, y\rangle = \langle Ax, Ay\rangle$
  $A\langle\langle x, y\rangle\rangle = \langle\langle Ax, Ay\rangle\rangle$
  (However, $Ax \cdot Ay \neq A(x \cdot y)$, because the resulting dimensions do not match. Also, $A(x \odot y) \neq Ax \odot Ay$)

- **NOT Commutative:**
  $Ax \neq xA$
  $AB \neq BA$

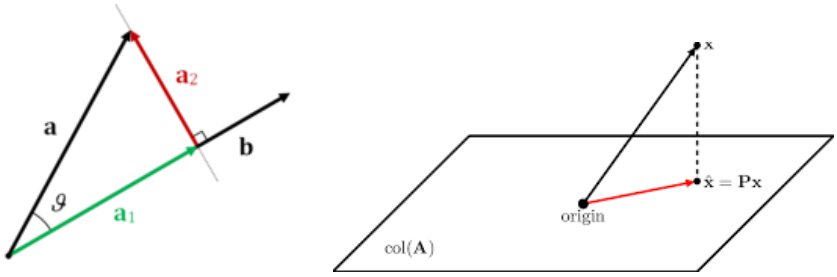
Proof.

The properties described in **??** can be demonstrated by expanding the formulas on both sides of each equation using a variable representation for each element in the vectors/matrices and comparing the resulting formulas. We leave this expansion as an exercise for the reader.

□

A-10.4 Projection

There are two types of projections: a vector projection and an orthogonal (i.e., plane) projection.



(a) Vector Projection (Source) (b) Orthogonal (Plane/Subspace) Projection (Source)

Figure 4

Vector Projection: Given two vectors $\vec{a}$ and $\vec{b}$ in the same n-dimensional vector space, the vector projection $\text{Proj}_{\vec{b}}(\vec{a})$ measures how much $\vec{a}$ contains the element of $\vec{b}$ (i.e., filtering $\vec{a}$ by $\vec{b}$). In the example of **??**, $\vec{a}$'s projection on $\vec{b}$ is $\vec{a}_1$, where the length of $\vec{a}_1$ is geometrically $||a_1|| = ||a|| \cos\theta = ||a|| \dfrac{a \cdot b}{||a|| \cdot ||b||} = \dfrac{a \cdot b}{||b||}$. Let $\vec{b'}$ be a unit vector of $\vec{b}$, that is $\vec{b'} = \dfrac{\vec{b}}{||b||}$. Then, $\vec{a}_1 = ||a_1|| \cdot \vec{b'} = \dfrac{a \cdot b}{||b||} \cdot \dfrac{\vec{b}}{||b||} = \dfrac{a \cdot b}{||b||^2}\vec{b}$. Thus, $\text{Proj}_{\vec{b}}(\vec{a}) = \dfrac{a \cdot b}{||b||^2}\vec{b}$.

Orthogonal Projection: Given the vector $\vec{x}$ and a set of mutually orthogonal vectors $\vec{p}_0, \vec{p}_1, \cdots, \vec{p}_{n-1}$ that span the plane (or subspace) P, the orthogonal projection $\text{Proj}_P(\vec{x})$ measures how much of the element of the plane P is contained in $\vec{x}$ (i.e., filtering $\vec{x}$ by the plane P). In the example of **??**, vector $\vec{x}$'s projection onto plane P is shown as a red arrow, which is computed by summing the projections of $\vec{x}$ onto each of the mutually orthogonal vectors $\vec{p}_0, \vec{p}_1, \cdots, \vec{p}_{n-1}$ that span plane P. This is equivalent to $\text{Proj}_P(\vec{a}) = \sum_{i=0}^{n-1} \text{Proj}_{\vec{p}_i}(\vec{a})$. The computation of $\text{Proj}_P(\vec{x})$ can be thought of as transforming $\vec{v}$ into a different coordinate system that expresses the vector space in terms of n mutually orthogonal vectors.

⟨Definition ??⟩ Vector and Orthogonal Projections

- **Vector Projection:** Given two vectors $\vec{a}$ and $\vec{b}$ in the same vector space, the vector projection of $\vec{a}$ on $\vec{b}$ is:
  $\text{Proj}_{\vec{b}}(\vec{a}) = \vec{a}_p = \vec{a}\cos\theta = \dfrac{\vec{a} \cdot \vec{b}}{||b||^2}\vec{b}$
- **Orthogonal Basis:** If the n-dimensional plane (or subspace) P is spanned by the mutually orthogonal n-dimensional vectors $\vec{p}_0, \vec{p}_1, \cdots, \vec{p}_n$,

then the matrix $P = \begin{bmatrix} \vec{p}_0 \\ \vec{p}_1 \\ \vdots \\ \vec{p}_{n-1} \end{bmatrix}$ is defined to be an orthogonal basis of plane P.

- **Orthogonal Projection:** Given the orthogonal basis matrix $P = \begin{bmatrix} \vec{p}_0 \\ \vec{p}_1 \\ \vdots \\ \vec{p}_{n-1} \end{bmatrix}$,

  vector $\vec{a}$'s orthogonal projection on P is:
  $\mathsf{Proj}_P(\vec{a}) = \sum_{i=1}^{n} \mathsf{Proj}_{\vec{p}_i}(\vec{a})$

Based on the definition of orthogonal projection, the following properties are derived:

Orthogonal Basis: In an n-dimensional vector space, any mutually orthogonal n vectors in the vector space span a plane (or subspace) P that is identical to the entire vector space. Further, the orthogonal projection of any vector in the vector space on P is guaranteed to be a unique vector.

Non-orthogonal Basis: In an n-dimensional vector space, suppose some n non-orthogonal vectors satisfy the following two conditions: (i) they span the entire vector space; (ii) they are linearly independent (i.e., one vector cannot be expressed as a linear combination of the other vectors). Then, the $n \times n$ matrix P comprised of these n vectors forms a basis for the entire vector space V, and the matrix-to-vector multiplication $P\vec{v}$ for each $\vec{v}$ in the vector space is guaranteed to yield a unique vector. However, the formula $\mathsf{Proj}_P(\vec{v})$ is not a valid geometric projection of the vector $\vec{v}$ on P, because the n basis vectors are non-orthogonal. Yet, the computation of $P\vec{v}$ can be thought of as uniquely transforming $\vec{v}$ into a different coordinate system that expresses the vector space with respect to n non-orthogonal vectors in P.

⟨Theorem ??⟩ Uniqueness of Transformed Vectors

- **Orthogonal Basis:** If some n vectors are a orthogonal basis of the plane P in the n-dimensional vector space, then P is the same as the entire vector space, and $\mathsf{Proj}_P(v)$ for every vector $\vec{v}$ in the vector space is guaranteed to be a unique vector.
- **Non-orthogonal Basis:** If some n vectors are a non-orthogonal basis of the plane P in the n-dimensional vector space (i.e., each vector is linearly independent and they span P), then P is the same as the entire vector space, and $P\vec{v}$ is guaranteed to result in a unique vector.

A-10.5 Basis of a Polynomial Ring

Given an $(n-1)$-degree polynomial ring $\mathbb{Z}[X]/(X^n+1)$, a basis of the polynomial ring is defined as a set of polynomials that satisfies the following two requirements:

- **Linear Independence**: Each polynomial in the basis set cannot be expressed as a linear combination of the other polynomials in the same set
- **Spanning the Polynomial Ring:** A linear combination of the polynomials in the basis set

can express any polynomial in the polynomial ring

Note that for a $(n-1)$-degree polynomial ring, the number of polynomials that form a basis of the polynomial ring is exactly n.

A-10.6 Isomorphism between Polynomials and Vectors over Integers

Now, let's define a mapping σ from the $(n-1)$-degree polynomial ring to the n-dimensional vector space, such that an input polynomial's list of y values evaluated at n distinct x coordinates (e.g., $x_0, x_1, \cdots, x_{n-1}$) forms the mapping's output vector. Technically, σ is defined as:

$\sigma : f(x) \in \mathbb{Z}[X]/(X^n+1) \longrightarrow (f(x_0), f(x_1), f(x_2), \cdots, f(x_{n-1})) \in \mathbb{Z}^n$

Now, we will explain why the mapping σ is isomorphic, which means that σ is a bijective one-to-one mapping from $\mathbb{Z}[X]/(X^n+1)$ to $\mathbb{Z}^n$, and the algebraic operations $(+, \cdot)$ of the mapped elements preserve correctness with homomorphism.

Bijective: In the $(n-1)$-degree polynomial ring, a list of y values evaluated at some statically chosen n distinct x coordinates defines a unique polynomial because, algebraically, there exists only one $(n-1)$-degree (or a lesser degree) polynomial that passes through each given set of n distinct (x, y) coordinates. We proved this in Lagrange Polynomial Interpolation (Theorem **??** in **??**).

Homomorphic: The homomorphism of the mapping σ on the $(+, \cdot)$ operations means that the following two relationships hold:

$\sigma(f_a(X) + f_b(X)) = \sigma(f_a(X)) + \sigma(f_b(X))$

$\sigma(f_a(X) \cdot f_b(X)) = \sigma(f_a(X)) \odot \sigma(f_b(X))$ # $\odot$ is Hadamard vector multiplication (Summary **??**)

To prove our σ mapping's homomorphism, let's denote the input polynomials $f_a(X)$, $f_b(X)$, and their σ-mapped output vectors as follows:

$f_a(X) = a_0 + a_1 X + a_2 X^2 + \cdots + a_{n-1} X^{n-1}$

$\sigma(f_a(X)) = (f_a(x_0), f_a(x_1), f_a(x_2), \cdots, f_a(x_{n-1})) = \left(\sum_{i=0}^{n-1} a_i (x_0)^i, \sum_{i=0}^{n-1} a_i (x_1)^i, \sum_{i=0}^{n-1} a_i (x_2)^i, \cdots, \sum_{i=0}^{n-1} a_i (x_{n-1})^i\right)$

$f_b(X) = b_0 + b_1 X + b_2 X^2 + \cdots + b_{n-1} X^{n-1}$

$\sigma(f_b(X)) = (f_b(x_0), f_b(x_1), f_b(x_2), \cdots, f_b(x_{n-1})) = \left(\sum_{i=0}^{n-1} b_i (x_0)^i, \sum_{i=0}^{n-1} b_i (x_1)^i, \sum_{i=0}^{n-1} b_i (x_2)^i, \cdots, \sum_{i=0}^{n-1} b_i (x_{n-1})^i\right)$

Given the above setup, we can see that σ preserves homomorphism on the $(+)$ operation as follows:

$$\boldsymbol{\sigma}(f_a(X) + f_b(X)) = \boldsymbol{\sigma}\Big((a_0+b_0) + (a_1+b_1)X + (a_2+b_2)X^2 + \cdots + (a_{n-1}+b_{n-1})X^{n-1}\Big)$$

$$= \left(\sum_{i=0}^{n-1} (a_i+b_i)(x_0)^i, \sum_{i=0}^{n-1} (a_i+b_i)(x_1)^i, \sum_{i=0}^{n-1} (a_i+b_i)(x_2)^i, \cdots, \sum_{i=0}^{n-1} (a_i+b_i)(x_{n-1})^i\right)$$

$$= \left(\sum_{i=0}^{n-1} a_i (x_0)^i, \sum_{i=0}^{n-1} a_i (x_1)^i, \sum_{i=0}^{n-1} a_i (x_2)^i, \cdots, \sum_{i=0}^{n-1} a_i (x_{n-1})^i\right)$$

$$+ \left(\sum_{i=0}^{n-1} b_i (x_0)^i, \sum_{i=0}^{n-1} b_i (x_1)^i, \sum_{i=0}^{n-1} b_i (x_2)^i, \cdots, \sum_{i=0}^{n-1} b_i (x_{n-1})^i\right)$$

$$= \boldsymbol{\sigma}(f_a(X)) + \boldsymbol{\sigma}(f_b(X))$$

Also, we can see that σ preserves homomorphism on the $(\cdot)$ operation as follows:

$$\boldsymbol{\sigma}(f_a(X) \cdot f_b(X)) = \boldsymbol{\sigma}\left(\left(\sum_{i=0}^{n-1} a_i X^i\right) \cdot \left(\sum_{i=0}^{n-1} b_i X^i\right) \right)$$

$$= \left(\left(\sum_{i=0}^{n-1} a_i x_0^i\right)\left(\sum_{i=0}^{n-1} b_i x_0^i\right), \ \left(\sum_{i=0}^{n-1} a_i x_1^i\right)\left(\sum_{i=0}^{n-1} b_i x_1^i\right), \ \left(\sum_{i=0}^{n-1} a_i x_2^i\right)\left(\sum_{i=0}^{n-1} b_i x_2^i\right), \right.$$

$$\left. \cdots, \left(\sum_{i=0}^{n-1} a_i x_{n-1}^i\right)\left(\sum_{i=0}^{n-1} b_i x_{n-1}^i\right) \right)$$

$$= \left(\sum_{i=0}^{n-1} a_i (x_0)^i, \ \sum_{i=0}^{n-1} a_i (x_1)^i, \ \cdots, \ \sum_{i=0}^{n-1} a_i (x_{n-1})^i \right)$$

$$\odot \left(\sum_{i=0}^{n-1} b_i (x_0)^i, \ \sum_{i=0}^{n-1} b_i (x_1)^i, \ \cdots, \ \sum_{i=0}^{n-1} b_i (x_{n-1})^i \right)$$

$$= \boldsymbol{\sigma}(f_a(X)) \odot \boldsymbol{\sigma}(f_b(X))$$

In summary, σ preserves the following homomorphism:
$\sigma(f_a(X) + f_b(X)) = \sigma(f_a(X)) + \sigma(f_b(X))$
$\sigma(f_a(X) \cdot f_b(X)) = \sigma(f_a(X)) \odot \sigma(f_b(X))$

However, for $\sigma(f_a(X) \cdot f_b(X)) = \sigma(f_a(X)) \odot \sigma(f_b(X))$, we need further reasoning to justify that this relation holds in polynomial rings, which is explained below.

Polynomial Ring Reduction: Suppose that we did not have the polynomial ring setup $X^n + 1$. Then, if we multiply $f_a(X)$ and $f_b(X)$, then $f_a(X) \cdot f_b(X)$ may become a new polynomial whose degree is higher than $n - 1$. This higher-degree polynomial would still decode into the expected correct vector. Suppose the following:
$\sigma(f_a(X)) = (f_a(x_0), f_a(x_1), \cdots, f_a(x_{n-1})) = (v_0, v_1, \cdots, v_{n-1})$
$\sigma(f_b(X)) = (f_b(x_0), f_b(x_1), \cdots, f_b(x_{n-1})) = (u_0, u_1, \cdots, u_{n-1})$

Then, the following is true:
$\sigma(f_a(X) \cdot f_b(X)) = (f_a(x_0) \cdot f_b(x_0), f_a(x_1) \cdot f_b(x_1), \cdots, f_a(x_{n-1}) \cdot f_b(x_{n-1})) = (v_0 u_0, v_1 u_1, \cdots, v_{n-1} u_{n-1})$
$= (v_0, v_1, \cdots, v_{n-1}) \odot (u_0, u_1, \cdots, u_{n-1})$

As shown above, even if $f_a(X) \cdot f_b(X)$ results in a polynomial with a degree higher than $n - 1$, it can be decoded into the expected correct vector. However, the σ mapping loses the property of isomorphism between a polynomial and a vector because if a polynomial's degree is higher than $n - 1$, then there can be more than 1 polynomial that passes through the given n distinct X coordinates: $\{x_0, x_1, \cdots, x_{n-1}\}$. This is a problem because, if the σ mapping supports only polynomial-to-vector mappings and not vector-to-polynomial mappings, then we cannot convert vectors into polynomials in the first place and do isomorphic computations. Another minor issue is that if the polynomial degree term is higher than $n - 1$, then the computational overhead of decoding (i.e., polynomial evaluation) becomes larger than before.

To resolve these two minor issues, we let the n distinct X coordinates of evaluation be the solutions of the polynomial ring modulo $X^n + 1$ (where n is some power of 2) and reduce $f_a(X) \cdot f_b(X)$

to a new polynomial modulo X^n+1 whose degree is at most $n-1$. Let $f_{ab}(X) = f_a(X) \cdot f_b(X)$, and $f'_{ab}(X)$ be the reduced polynomial such that $f_{ab}(X) = Q(X) \cdot (X^n+1) + f'_{ab}(X)$ for some quotient polynomial $Q(X)$. Then, as illustrated in Summary **??** (**??**), $f_{ab}(X)$ and $f'_{ab}(X)$ evaluate to the same value if they are evaluated at the roots of X^n+1 (by zeroing out the $Q(X)$ term). Therefore, if we let the n distinct evaluating points $\{x_0, x_1, \cdots, x_{n-1}\}$ be the roots of X^n+1, then we can ensure that the decoded vector of $f'_{ab}(X)$ is identical to that of $f_{ab}(X)$, which we expect. Therefore, we can replace the higher-degree polynomial $f_{ab}(X)$ with the reduced polynomial $f'_{ab}(X)$ and continue with any further polynomial additions or multiplications using $f'_{ab}(X)$. Also, by applying polynomial ring reduction, we can enhance the computational efficiency of polynomial addition and multiplication, as well as preserve the isomorphism of the σ mapping. Therefore, we can freely convert between vectors & polynomials and perform additions and multiplications.

For applying this polynomial ring reduction, the polynomial modulus can be any polynomial as long as it has at least n distinct roots. In practice, we often choose X^n+1 as the polynomial ring modulus, which is the $(\mu = 2n)$-th cyclotomic polynomial (**??**). The reason we let the polynomial ring modulus be a cyclotomic polynomial (especially the $(\mu = 2n)$-th cyclotomic polynomial, X^n+1) is that its n distinct roots are well-defined (i.e., primitive $(\mu = 2n)$-th roots of unity) and thus can be quickly computed even when n is large.

Polynomial Coefficient Modulo Reduction: In addition, we often reduce the polynomial coefficients based on some modulus t to keep the size of the coefficients lower than a certain limit for the purpose of computational efficiency. Suppose two polynomials $f_c(X)$ and $f_d(X)$ have congruent coefficients modulo t as follows:

$f_c(X) = w_0 + w_1 \cdot x_i + w_2 \cdot x_i^2 + \cdots + w_{n-1} \cdot x_i^{n-1}$
$f_d(X) = w'_0 + w'_1 \cdot x_i + w'_2 \cdot x_i^2 + \cdots + w'_{n-1} \cdot x_i^{n-1}$
$w_i \equiv w'_i \bmod t$

Then, their evaluated value $f_c(x_i)$ and $f_d(x_i)$ for any x_i is guaranteed to be congruent modulo t, as shown below:

$f_c(x_i) = w_0 + w_1 \cdot x_i + w_2 \cdot x_i^2 + \cdots + w_{n-1} \cdot x_i^{n-1}$
$\equiv (w_0 + w_1 \cdot x_i + w_2 \cdot x_i^2 + \cdots + w_{n-1} \cdot x_i^{n-1}) \bmod t$
$\equiv (w_0 \bmod t) + (w_1 \bmod t) \cdot x_i + (w_2 \bmod t) \cdot x_i^2 + \cdots + (w_{n-1} \bmod t) \cdot x_i^{n-1})$
$\equiv w'_0 + w'_1 \cdot x_i + w'_2 \cdot x_i^2 + \cdots + w'_{n-1} \cdot x_i^{n-1}$
$= f_d(x_i)$

Summary: Since σ is bijective and homomorphic, σ is an isomorphic mapping between the $(n-1)$-degree polynomial ring $\mathbb{Z}_t[X]/X^n+1$ and the n-dimensional vector space $\mathbb{Z}_t^n$.

A-10.6.1 Finding Appropriate Modulus t

To isomorphically evaluate a polynomial in $\mathbb{Z}_t[X]/X^n+1$ into an n-dimensional vector, we need to evaluate the polynomial at n distinct roots of $X^n+1 \bmod t$. However, $X^n+1 \bmod t$ does not have n distinct roots for all combinations of (degree, modulus) $= (n, t)$. For example, if $n = 2$ and $t = 3$, then $X^2+1 \not\equiv 0 \bmod 3$ for any possible values of $X = \{0, 1, 2\}$. Therefore, our goal is to find a satisfactory t given a fixed n such that n distinct roots of $X^n+1 \pmod t$ exist, in order to use the isomorphic σ mapping.

We start with two constraints: (1) we choose t only such that t is a prime number p; (2) $t1$ is some multiple of $2n$ (i.e., $(t1) = k \cdot 2n$ for some integer k).

We learned from Fermat's Little Theorem in Theorem **??**.4 (**??**) the following: $a^{t-1} \equiv 1 \bmod t$ if and only if a and t are co-prime. This means that if t is a prime, then $a^{t-1} \equiv 1 \bmod t$ for all $a \in \mathbb{Z}_t^\times$ (i.e., $\mathbb{Z}_t$ without $\{0\}$). Suppose g is the generator of $\mathbb{Z}_t^\times$ whose powered values generate all elements of $\mathbb{Z}_t^\times$. Then, $\mathsf{Ord}_{\mathbb{Z}_t}(g) = t-1$ and $g^{t-1} \equiv 1 \bmod t$. Since $t-1 = k \cdot 2n$ for some k, $g^{k \cdot 2n} \equiv (g^k)^{2n} \equiv 1 \bmod t$. Then, $\mathsf{Ord}_{\mathbb{Z}_t}(g^k) \leq 2n$. However, since $\mathsf{Ord}_{\mathbb{Z}_t}(g) = t-1$, for all a such that $a < t1 = k \cdot 2n$, $g^a \not\equiv 1 \bmod t$. In other words, for all b such that $b < 2n$, $(g^k)^b \not\equiv 1 \bmod t$. Thus, $\mathsf{Ord}_{\mathbb{Z}_t}(g^k) = 2n$.

Let $c = g^k$. Since $\mathsf{Ord}_{\mathbb{Z}_t}(c) = 2n$, $c^{2n} \equiv 1 \bmod t$. In other words, $(c^n)^2 \equiv 1 \bmod t$. Now, c^n can be only 1 or -1. The reason is that in the relation $X^2 \equiv 1 \bmod t$, X can be mathematically only 1 or $-1 \equiv t1 \bmod t$. If we substitute $X = c^n$, then c^n can be only 1 or $-1 \equiv t-1 \bmod t$. But $\mathsf{Ord}_{\mathbb{Z}_t}(c) = 2n$, thus c^n cannot be 1 (because $1^1 = 1$ and 1 is smaller than the order of c: $2n > 1$). Thus, c^n can be only $-1 \equiv t1 \bmod t$. If $c^n = -1 \equiv t-1 \bmod t$, then c is the root of $X^n + 1 \bmod t$, because $X^n + 1 = c^n + 1 \equiv (t-1) + 1 \equiv 0 \bmod t$.

In conclusion, given a cyclotomic polynomial $X^n + 1$, if we choose a prime t such that $t1 = k \cdot 2n$ for some integer k, then one root of $X^n + 1 \bmod t$ is: $X = c = g^k$.

Once we have found one root of X^n+1, then we can derive all n distinct roots of X^n+1. Suppose ω is one root of $X^n + 1 \bmod t$. Then, we derive the following:

$\omega^n + 1 \equiv 0 \bmod t$

$\omega^n \equiv t-1 \bmod t$

$\omega^{2n} \equiv (t-1) \cdot (t-1) \equiv t^2 2t + 1 \equiv 1 \bmod t$

While $\omega^{2n} \equiv 1 \bmod t$, $\omega^n \not\equiv 1 \bmod t$, because if so, $X^n + 1 = \omega^n + 1 = 1 + 1 = 2 \neq 0$, which contradicts the fact that ω is a root of $X^n + 1$. Therefore, $\mathsf{Ord}_{\mathbb{Z}_t}(c) = 2n$.

Now, we derive the remaining $n-1$ distinct roots of $X^n + 1 \bmod t$ as follows:

$(\omega^3)^n + 1 \equiv (\omega^{2n}) \cdot \omega^n + 1 \equiv \omega^n + 1 \equiv t - 1 + 1 \equiv 0 \bmod t$

$(\omega^5)^n + 1 \equiv (\omega^{4n}) \cdot \omega^n + 1 \equiv \omega^n + 1 \equiv t - 1 + 1 \equiv 0 \bmod t$

$\vdots$

$(\omega^{2n-1})^n + 1 \equiv (\omega^{2n-2}) \cdot \omega^n + 1 \equiv \omega^n + 1 \equiv t - 1 + 1 \equiv 0 \bmod t$

Note that $\omega, \omega^3, \omega^5, \cdots, \omega^{2n-1}$ are all distinct values in $\mathbb{Z}_t^\times$, because $\mathsf{Ord}_{\mathbb{Z}_t}(c) = 2n$. Thus, $\{\omega^{2i+1}\}_{i=0}^{n-1}$ are n distinct roots of $X^n + 1$. At the same time, since these are the roots of the cyclotomic polynomial $X^n + 1$, these are n distinct primitive ($\mu = 2n$)-th roots of unity.

We summarize our findings as follows:

⟨Theorem ??⟩ Isomorphism between Polynomials and Vectors over Integer (Ring)

- Suppose we have an $(n-1)$-degree polynomial ring $\mathbb{Z}_t[X]/F(X)$ where $F(X)$ is an n-degree polynomial having n distinct roots $\{x_0, \cdots, x_{n-1}\}$, and we have an n-dimensional modulo p vector space $\vec{v} \in \mathbb{Z}_t$. We define mapping σ between this polynomial ring and vector space as follows:
  $\sigma : f(x) \in \mathbb{Z}_t[X]/F(X) \longrightarrow (f(x_0), f(x_1), f(x_2), \cdots, f(x_{n-1})) \in \mathbb{Z}_t^n$

  Then, σ preserves isomorphism over the $(+, \cdot)$ operations:

$\sigma(f_a(X) + f_b(X)) = \sigma(f_a(X)) + \sigma(f_b(X))$
$\sigma(f_a(X) \cdot f_b(X)) = \sigma(f_a(X)) \odot \sigma(f_b(X))$

- If the n-degree polynomial $F(x)$ has a fewer number of roots than n, then isomorphism between the vectors and polynomials over the $(+, \cdot)$ operations holds without modulo t as follows:
  $\sigma' : f(x) \in \mathbb{Z}[X]/F(X) \longrightarrow (f(x_0), f(x_1), f(x_2), \cdots, f(x_{n-1})) \in \mathbb{Z}^n$

- Suppose we have the $(\mu = 2n)$-th cyclotomic polynomial $X^n + 1 \bmod t$ such that t is a prime and $t - 1$ is some multiple of $2n$, and g is a generator of $\mathbb{Z}_t^\times$. Then, n distinct roots of $X^n + 1$ (i.e., primitive $(\mu = 2n)$-th roots of unity) can be efficiently computed as: $\{\omega^{2i+1}\}_{i=0}^{n-1}$ where $\omega = g^{\frac{t-1}{2n}} \bmod t$.

A-10.7 Isomorphism between Polynomials and Vectors over Complex Numbers

In Theorem **??** (**??**), we learned the isomorphic mapping $\sigma : f(X) \in \mathbb{Z}_t[X]/(X^n+1) \longrightarrow (f(x_0), f(x_1), f(x_2), \cdots, f($ $\mathbb{Z}_t^n$, where $x_0, x_1, x_2, \cdots, x_{n-1}$ are the ($(\mu = 2n)$-th primitive) roots of the cyclotomic polynomial $X^n + 1$, which are $\omega, \omega^3, \omega^5, \cdots, \omega^{2n-1}$, where ω can be any root of $X^n + 1$ (i.e., since each ω is a generator of all roots). In this subsection, we will demonstrate the isomorphism between a vector space and a polynomial ring over complex numbers as follows:

$\sigma_c : f(X) \in \mathbb{R}[X]/(X^n + 1) \longrightarrow (f(\omega), f(\omega^3), f(\omega^5), \cdots, f(\omega^{2n-1})) \in \hat{\mathbb{C}}^n$

, where $\omega = e^{i\pi/n}$, the root (i.e., the primitive $(\mu = 2n)$-th root) of the cyclotomic polynomial $X^n + 1$ over complex numbers (Theorem **??**.1 in **??**). We define $\hat{\mathbb{C}}^n$ to be an n-dimensional *special* vector space whose second-half elements of each vector are reverse-ordered conjugates of the first-half elements (e.g., $(v_0, v_1, \cdots, v_{\frac{n}{2}-1}, \overline{v}_{\frac{n}{2}-1}, \cdots, \overline{v}_1, \overline{v}_0)$).

A-10.7.1 Isomorphism between $\mathbb{C}^{\frac{n}{2}}$ and $\hat{\mathbb{C}}^n$

Bijective: Technically, $\hat{\mathbb{C}}^n$ is bijective to $\mathbb{C}^{\frac{n}{2}}$, because the second-half $\frac{n}{2}$ elements of each vector in $\hat{\mathbb{C}}^n$ are passively (automatically) determined by the first-half $\frac{n}{2}$ elements. Therefore, each vector in $\hat{\mathbb{C}}^n$ has one-to-one correspondences with some unique vector in $\mathbb{C}^{\frac{n}{2}}$, and thus these two vector spaces are bijective.

Homomorphic: To demonstrate their homomorphism over the $(+, \odot)$ operations, we can apply the following reasoning: for all $\vec{\hat{v}} \in \hat{\mathbb{C}}^n$ and $\vec{v} \in \mathbb{C}^{\frac{n}{2}}$, there exists an $\frac{n}{2} \times n$ linear transformation matrix M that satisfies $M \cdot \vec{\hat{v}} = \vec{v}$. Such M is an $\frac{n}{2} \times n$ matrix comprising horizontal concatenation of $I_{\frac{n}{2}}$ and $[0]_{\frac{n}{2}}$, where $I_{\frac{n}{2}}$ is an $\frac{n}{2} \times \frac{n}{2}$ identity matrix and $[0]_{\frac{n}{2}}$ is an $\frac{n}{2} \times \frac{n}{2}$ zero matrix. Also, there exists an $n \times \frac{n}{2}$ (non-linear) transformation matrix N that satisfies $N \cdot \vec{v} = \vec{\hat{v}}$. Such N is a vertical concatenation of $I_{\frac{n}{2}}$ and $\emptyset^R_{\frac{n}{2}}$, where $\emptyset^R_{\frac{n}{2}}$ is an $\frac{n}{2} \times \frac{n}{2}$ matrix whose reverse-diagonal elements

are unary conjugate operators and all other elements are zero. For example, if $n = 4$, then M and N are structured as follows:

$$M = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \end{bmatrix}, \quad N = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & \emptyset \\ 0 & 0 & \emptyset & 0 \\ 0 & \emptyset & 0 & 0 \\ \emptyset & 0 & 0 & 0 \end{bmatrix}$$

The reason N is not a linear transformation matrix is because it contains conjugate operators $\emptyset$. Yet, notice that the following homomorphism holds between $\vec{v} \in \hat{\mathbb{C}}^n$ and $\vec{v} \in \mathbb{C}^{\frac{n}{2}}$:

$N \cdot (M \cdot \vec{v}_1 + M \cdot \vec{v}_2) = \vec{v}_1 + \vec{v}_2, \quad M \cdot (N \cdot \vec{v}_1 + N \cdot \vec{v}_2) = \vec{v}_1 + \vec{v}_2$

$N \cdot (M \cdot \vec{v}_1 \odot M \cdot \vec{v}_2) = \vec{v}_1 \odot \vec{v}_2, \quad M \cdot (N \cdot \vec{v}_1 \odot N \cdot \vec{v}_2) = \vec{v}_1 \odot \vec{v}_2$

Thus, the $\hat{\mathbb{C}}^n$ and $\mathbb{C}^{\frac{n}{2}}$ vector spaces are bijective and homomorphic over the $(+, \odot)$ operations, and therefore they preserve isomorphism.

A-10.7.2 Isomorphism between $\hat{\mathbb{C}}^n$ and $\mathbb{R}[X]/X^n + 1$

Now, we will demonstrate σ_c's isomorphism (i.e., bijective and homomorphic) between $\hat{\mathbb{C}}^n$ and $\mathbb{R}[X]/X^n + 1$ by applying the same reasoning as described in the beginning of **??**.

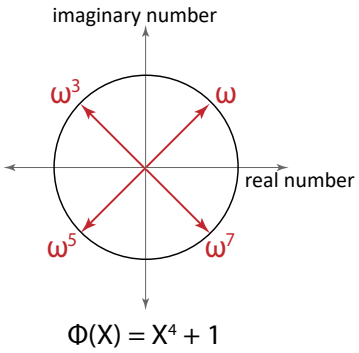


Figure 5: An illustration of the four roots of the 8th cyclotomic polynomial $x^4 + 1$

Bijective: Based on Euler's formula $e^{i\theta} = \cos\theta + i \cdot \sin\theta$ (**??**), we can derive the following arithmetic relations: $\omega = \overline{\omega^{2n-1}}$, $\omega^3 = \overline{\omega^{2n-3}}, \cdots, \omega^{n-1} = \overline{\omega^{n+1}}$. In other words, the one-half roots are conjugates of the other-half roots. This can also be pictorially understood based on a complex plane in **??**, where red arrows represent the roots of the 8th cyclotomic polynomial $X^4 + 1$, comprising imaginary number and real number components. As shown in this figure, one half of the red arrows (i.e., roots) are a reflection of the other half on the x-axis (i.e., imaginary number axis). This means that we can express these roots as an n-dimensional vector whose elements are the roots of $X^n + 1$, such that its second-half elements are a reverse-ordered conjugate of the first-half elements. Based on this vector design, the σ mapping can be re-written as follows:

$\sigma(f(X)) = (f(\omega), f(\omega^3), f(\omega^5), \cdots, f(\omega^{n-1}), f(\overline{\omega^{n-1}}), f(\overline{\omega^{n-3}}), \cdots, f(\overline{\omega^3}), f(\overline{\omega}))$

Since $f(\overline{X}) = \overline{f(X)}$, we can rewrite σ as:
$\sigma(f(X)) = (f(\omega), f(\omega^3), f(\omega^5), \cdots, f(\omega^{n-3})f(\omega^{n-1}), \overline{f(\omega^{n-1})}, \overline{f(\omega^{n-3})}, \cdots, \overline{f(\omega^3)}, \overline{f(\omega)})$

This structure of vector exactly aligns with the definition of $\hat{\mathbb{C}}^n$: the first half of the elements of the n-dimensional vector $\vec{v}$ is a conjugate of the second half.

For bijectiveness, we also need to demonstrate that every $f(X) \in \mathbb{R}[X]/X^n + 1$ is mapped to some $\vec{v} \in \hat{\mathbb{C}}^{\frac{n}{2}}$, and no two different $f_1(X), f_2(X) \in \mathbb{R}[X]/X^n + 1$ map to the same $\vec{v} \in \hat{\mathbb{C}}^n$. The first requirement is satisfied because each polynomial $f(X) \in \mathbb{R}[X]/X^n + 1$ can be evaluated at the n distinct roots of $X^n + 1$ to a valid number. The second requirement is also satisfied because in the $(n-1)$-degree polynomial ring, each list of n distinct (x, y) coordinates (where we fix the X values to the n distinct roots of $X^2 + 1$ as $\{\omega, \omega^3, \cdots, \omega^{2n-1}\}$) can be mapped only to a single polynomial within the $(n-1)$-degree polynomial ring, as proved by Lagrange Polynomial Interpolation (Theorem **??** in **??**).

Homomorphic: σ_c is homomorphic, because based on the reasoning shown in **??**, the relations $\sigma(f_a(X) + f_b(X)) = \sigma(f_a(X)) + \sigma(f_b(X))$ and $\sigma(f_a(X) \cdot f_b(X)) = \sigma(f_a(X)) \odot \sigma(f_b(X))$ mathematically hold regardless of whether the type of X is modulo integer or complex number,

Since σ_c is both bijective and homomorphic over the $(+, \cdot)$ operations, it is isomorphic.

⟨Theorem ??⟩ Isomorphism between Polynomials and Vectors over Complex Numbers

The following mapping σ_c between polynomials and vectors over complex numbers is isomorphic:
$\sigma_c : f(X) \in \mathbb{R}[X]/(X^n + 1) \longrightarrow (f(\omega), f(\omega^3), f(\omega^5), \cdots, f(\omega^{2n-1})) \in \hat{\mathbb{C}}^n \ (\longrightarrow \mathbb{C}^{\frac{n}{2}})$

, where $\omega = e^{i\pi/n}$, the root (i.e., the primitive ($\mu = 2n$)-th root) of the cyclotomic polynomial $X^n + 1$ over complex numbers, and $\hat{\mathbb{C}}^n$ is n-dimensional complex special vector space whose second-half elements are reverse-ordered conjugates of the first-half elements.

A-10.8 Transforming Basis between Polynomial Ring and Vector Space

Suppose some polynomials $f_0(X), f_1(X), \cdots, f_{n-1}(X)$ form a basis of the $(n-1)$-degree polynomial ring and σ is an isomorphic mapping from the $(n-1)$-degree polynomial ring to the n-dimensional vector space $\in \mathbb{Z}^n$. Then, $(\sigma(f_0(X)), \sigma(f_1(X)), \cdots, \sigma(f_{n-1}(X)))$ form a basis of the n-dimensional vector space. This is because the σ-mapped output vectors homomorphically preserve the same algebraic relationships on the $(+, \cdot)$ operations and the basis relationship between basis vectors and a subspace can be expressed as a linear algebraic formula consisting of the $(+, \cdot)$ operations (i.e., linear independence and spanning of the space). Therefore, if a set of polynomials satisfies a basis relationship, their σ-mapped vectors also preserve a basis relationship.

The same principle holds between a polynomial ring and vector space over complex numbers. Given the polynomial ring $\mathbb{R}[X]/(x^n + 1)$, the most intuitive way to set up a basis of $\mathbb{R}[X]/(x^n + 1)$

is as follows:

$f_0(X) = 1$
$f_1(X) = X$
$f_2(X) = X^2$
$\vdots$
$f_{n-1}(X) = X^{n-1}$

These n polynomials are linearly independent, because each polynomial exclusively has its own unique exponent term, whereas one term cannot be expressed by a linear combination of the other terms. Also, these n polynomials span the polynomial ring $\mathbb{R}[X]/(x^n+1)$, because each polynomial's scalar multiplication can express any coefficient value of its own exponent term, and summing all such polynomials can express any polynomial in the polynomial ring $\mathbb{R}[X]/(X^n+1)$.

Now, we will apply the σ_c mapping to the above n polynomials that are a basis of the $(n-1)$-degree polynomial ring $\mathbb{R}[X]/(x^n+1)$. Then, according to the principle of polynomial-to-vector basis transfer (explained in Theorem **??** in **??**), we can use these n polynomials (i.e., the basis of the $(n-1)$-degree polynomial ring) and the isomorphic polynomial-to-vector mapping σ_c to compute the basis of the n-dimensional special vector space $\hat{\mathbb{C}}^n$ as follows:

$$W = \begin{bmatrix} \sigma_c(f_0(X)) \\ \sigma_c(f_1(X)) \\ \sigma_c(f_2(X)) \\ \vdots \\ \sigma_c(f_{n-1}(X)) \end{bmatrix} = \begin{bmatrix} \sigma_c(1) \\ \sigma_c(X) \\ \sigma_c(X^2) \\ \vdots \\ \sigma_c(X^{n-1}) \end{bmatrix} = \begin{bmatrix} 1 & 1 & \cdots & 1 & 1 \\ (\omega) & (\omega^3) & (\omega^5) & \cdots & (\omega^{2n-1}) \\ (\omega)^2 & (\omega^3)^2 & (\omega^5)^2 & \cdots & (\omega^{2n-1})^2 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ (\omega)^{n-1} & (\omega^3)^{n-1} & (\omega^5)^{n-1} & \cdots & (\omega^{2n-1})^{n-1} \end{bmatrix}$$

$$= \begin{bmatrix} 1 & 1 & 1 & \cdots & 1 \\ (\omega) & (\omega^3) & \cdots & (\overline{\omega})^3 & (\overline{\omega}) \\ (\omega)^2 & (\omega^3)^2 & \cdots & (\overline{\omega}^3)^2 & (\overline{\omega})^2 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ (\omega)^{n-1} & (\omega^3)^{n-1} & \cdots & (\overline{\omega}^3)^{n-1} & (\overline{\omega})^{n-1} \end{bmatrix}$$

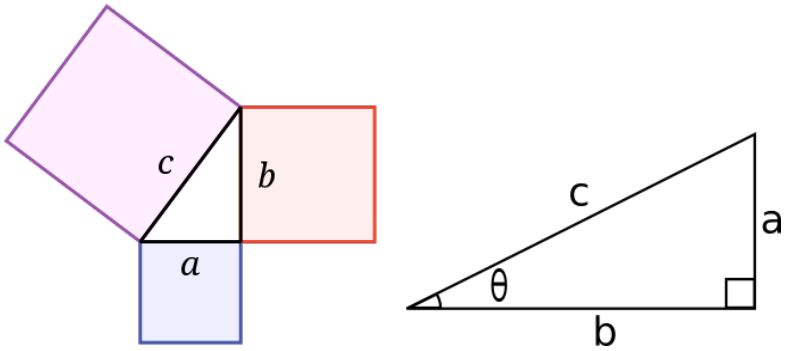
W is a valid basis of the n-dimensional special vector space $\hat{\mathbb{C}}^n$.

⟨Theorem ??⟩ Transforming Basis between Polynomial Ring and Vector Space

If n polynomials form a basis of an $(n-1)$-degree polynomial ring and they are converted into n distinct vectors via an isomorphic mapping σ (or σ_c in the case of the complex number domain) from the $(n-1)$-degree polynomial ring to the n-dimensional vector space, then those converted n vectors form a basis of the n-dimensional (or $\frac{n}{2}$ in the case of the complex number domain) vector space.

A-11 Euler's Formula

A-11.1 Pythagorean Identity



(a) Pythagorean Theorem (Source) (b) Pythagorean Identity (Source)

Figure 6

⟨Theorem ??⟩ Pythagorean Identity

$$\sin^2\theta + \cos^2\theta = 1$$

Proof.

1. In **??**, by Pythagorean theorem, $a^2 + b^2 = c^2$.
2. In **??**, by definition of sin and cosine,
   $\sin\theta = \frac{a}{c}, \quad \cos\theta = \frac{b}{c}$
3. Combining step 1 and 2:
   $\sin^2\theta + \cos^2\theta = \frac{a^2}{c^2} + \frac{b^2}{c^2} = \frac{a^2+b^2}{c^2} = 1$

□

A-11.2 Imaginary Number

⟨Definition ??⟩ Imaginary Number

- $\boldsymbol{i}$ is defined to be an imaginary number that has the property: $\boldsymbol{i^2 = -1}$
- **Complex Number** is a number that involves imaginary number(s) (e.g., $\boldsymbol{a = 5.6 + i \cdot 4.3}$)
- **Real Number** is a number that does not involve any imaginary number (e.g., $\boldsymbol{a = 13.4}$)
- $\overline{\boldsymbol{a}}$ is a **Conjugate** of $\boldsymbol{a}$ if $\boldsymbol{a}$ and $\overline{\boldsymbol{a}}$ have the same real number part and an opposite-signed imaginary number part
  (e.g., $\boldsymbol{a = 3 + i \cdot 3.4}$, $\quad \overline{\boldsymbol{a}} \boldsymbol{= 3 - i \cdot 3.4}$)
- **Hermitian Vector** is a vector where the 1st half of its elements is a conjugate of the 2nd half, as the $\boldsymbol{n}$-dimensional vector illustrated below:

$$\vec{v} = (v_1, v_2, v_3, \cdots, v_{\frac{n}{2}-1}, v_{\frac{n}{2}}, \overline{v}_{\frac{n}{2}}, \overline{v}_{\frac{n}{2}-1}, \cdots, \overline{v}_3, \overline{v}_2, \overline{v}_1)$$

A-11.3 Euler's Formula

⟨Definition ??⟩ Euler's Formula

$e^{i\theta} = \cos\theta + i \cdot \sin\theta$

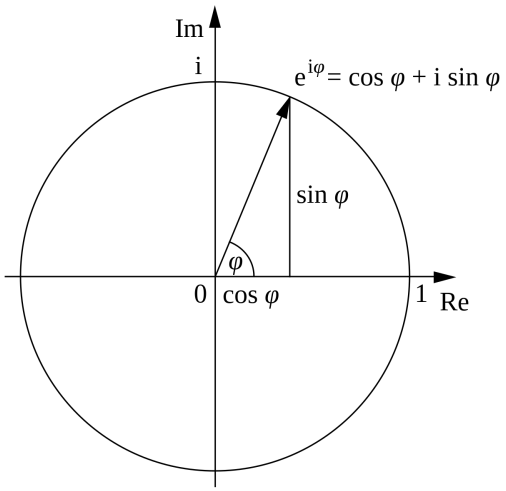


Figure 7: The figure illustrates a circle of Euler's formula in the complex plane (Source)

The value of $e^{i\theta}$ is represented as a coordinate on a circle in the complex plane in **??**, where the x-axis encodes the value's real number part and the y-axis encodes the value's imaginary number part. Note that as θ increases, the complex part oscillates between i and $-i$, and the real part oscillates between 1 and -1, with the period of 2π.

A-11.4 Vandermonde Matrix with Roots of Cyclotomic Polynomial over Complex Numbers

In this subsection, we will build a Vandermonde matrix (**??**) with the n distinct roots of the μ-th cyclotomic polynomial over complex numbers (where μ is a power of 2) as follows:

⟨Theorem ??⟩ Vandermonde Matrix with the Roots of (power-of-2)-th Cyclotomic Polynomial over Complex Numbers

Suppose we have an $n \times n$ (where n is a power of 2) Vandermonde matrix comprised of n distinct roots of the μ-th cyclotomic polynomial (explained in Theorem **??**.1 in **??**), where μ is a power of 2 and $n = \frac{\mu}{2}$. In other words, $V = Vander(x_0, x_1, \cdots, x_{n-1})$, where each $x_j = (e^{i\pi/n})^{2j-1}$ for $1 \leq j \leq n$ (i.e., the primitive μ-th roots of unity). Then, the following holds:

$$V \cdot V^T = \begin{bmatrix} 0 & \cdots & 0 & 0 & n \\ 0 & \cdots & 0 & n & 0 \\ 0 & \cdots & n & 0 & 0 \\ \vdots & \iddots & \vdots & \vdots & \vdots \\ n & 0 & 0 & \cdots & 0 \end{bmatrix} = n \cdot I_n^R$$

And $V^{-1} = \dfrac{V^T \cdot I_n^R}{n}$

Proof.

1. Given $\omega = e^{i\pi/n}$, each $x_j = (\omega)^{2j-1}$. Thus, we can expand as follows:

$$V \cdot V^T = \begin{bmatrix} 1 & (\omega) & (\omega)^2 & \cdots & (\omega)^{n-1} \\ 1 & (\omega^3) & (\omega^3)^2 & \cdots & (\omega^3)^{n-1} \\ 1 & (\omega^5) & (\omega^5)^2 & \cdots & (\omega^5)^{n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & (\omega^{2n-1}) & (\omega^{2n-1})^2 & \cdots & (\omega^{2n-1})^{n-1} \end{bmatrix} \cdot \begin{bmatrix} 1 & 1 & 1 & \cdots & 1 \\ (\omega) & (\omega^3) & (\omega^5) & \cdots & (\omega^{2n-1}) \\ (\omega)^2 & (\omega^3)^2 & (\omega^5)^2 & \cdots & (\omega^{2n-1})^2 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ (\omega)^{n-1} & (\omega^3)^{n-1} & (\omega^5)^{n-1} & \cdots & (\omega^{2n-1})^{n-1} \end{bmatrix}$$

$$= \begin{bmatrix} \sum_{k=0}^{n-1} \omega^{2k} & \sum_{k=0}^{n-1} \omega^{4k} & \sum_{k=0}^{n-1} \omega^{6k} & \cdots & \sum_{k=0}^{n-1} \omega^{2nk} \\ \sum_{k=0}^{n-1} \omega^{4k} & \sum_{k=0}^{n-1} \omega^{6k} & \sum_{k=0}^{n-1} \omega^{8k} & \cdots & \sum_{k=0}^{n-1} \omega^{2k(n+1)} \\ \sum_{k=0}^{n-1} \omega^{6k} & \sum_{k=0}^{n-1} \omega^{8k} & \sum_{k=0}^{n-1} \omega^{10k} & \cdots & \sum_{k=0}^{n-1} \omega^{2k(n+2)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \sum_{k=0}^{n-1} \omega^{2nk} & \sum_{k=0}^{n-1} \omega^{2(n+1)k} & \sum_{k=0}^{n-1} \omega^{2(n+2)k} & \cdots & \sum_{k=0}^{n-1} \omega^{2(n+n-1)k} \end{bmatrix}$$

2. The $V \cdot V^T$ matrix's anti-diagonal elements are $\sum_{k=0}^{n-1} \omega^{2nk}$. We can derive the following:

$$\sum_{k=0}^{n-1} \omega^{2nk} = \sum_{k=0}^{n-1} (e^{i\pi/n})^{2nk} = \sum_{k=0}^{n-1} e^{2\pi ki} = \sum_{k=0}^{n-1} (\cos(2\pi k) + i\sin(2\pi k)) = \sum_{k=0}^{n-1} (1+0) = n$$

This means that the The $V \cdot V^T$ matrix's anti-diagonal elements are n.

3. Next, we will prove that the $V \cdot V^T$ matrix has 0 for all positions except for the anti-diagonal ones. In other words, we will prove the following:

$$\sum_{k=0}^{n-1} \omega^{2k} = \sum_{k=0}^{n-1} \omega^{4k} = \sum_{k=0}^{n-1} \omega^{6k} = \cdots = \sum_{k=0}^{n-1} \omega^{2(n-1)k} = \sum_{k=0}^{n-1} \omega^{2(n+1)k} = \cdots = \sum_{k=0}^{n-1} \omega^{2(2n-1)k} = 0$$

For this proof, we will leverage the Geometric Sum formula $\sum_{i=0}^{n-1} x^i = \dfrac{x^n - 1}{x - 1}$:

⟨Theorem ??.1⟩ Geometric Sum Formula

Let the geometric sum $S_n = 1 + x + x^2 + \cdots + x^{n-1}$
Then, $x \cdot S_n = x + x^2 + x^3 + \cdots + x^n$
$x \cdot S_n - S_n = (x + x^2 + x^3 + \cdots + x^n) - (1 + x + x^2 + \cdots + x^{n-1}) = x^n - 1$

$S_n \cdot (x-1) = x^n - 1$
$S_n = \dfrac{x^n-1}{x-1}$ # with the constraint that $x \neq 1$

Leveraging the Geometric Sum formula $\sum_{i=0}^{n-1} x^i = \dfrac{x^n-1}{x-1}$,
$\sum_{k=0}^{n-1} \omega^{2mk} = \dfrac{(\omega^{2m})^n - 1}{\omega^{2m}-1} = \dfrac{(\omega^{2n})^m - 1}{\omega^{2m}-1} = \dfrac{1-1}{\omega^{2m}-1} = 0$ for $1 \leq m \leq (2n-1)$ # since $\mathsf{Ord}(\omega) = 2n$
Therefore,
$\sum_{k=0}^{n-1} \omega^{2k} = \sum_{k=0}^{n-1} \omega^{4k} = \sum_{k=0}^{n-1} \omega^{6k} = \cdots = \sum_{k=0}^{n-1} \omega^{2(n-1)k} = \sum_{k=0}^{n-1} \omega^{2(n+1)k} = \cdots = \sum_{k=0}^{n-1} \omega^{2(2n-1)k} = 0$

4. Based on the proof of step 2 and 3, $V \cdot V^T = \begin{bmatrix} 0 & \cdots & 0 & 0 & n \\ 0 & \cdots & 0 & n & 0 \\ 0 & \cdots & n & 0 & 0 \\ \vdots & \iddots & \vdots & \vdots & \vdots \\ n & 0 & 0 & \cdots & 0 \end{bmatrix} = n \cdot I_n^R$

5. Given $V \cdot V^T = n \cdot I_n^R$,
$V^{-1} \cdot V \cdot V^T = V^{-1} \cdot n \cdot I_n^R$
$V^T = V^{-1} \cdot n \cdot I_n^R$
$V^T \cdot I_n^R = V^{-1} \cdot n \cdot I_n^R \cdot I_n^R$
$V^T \cdot I_n^R = V^{-1} \cdot n$ # since $I_n^R \cdot I_n^R = I_n$
$V^{-1} = \dfrac{V^T \cdot I_n^R}{n}$

□

Later in the CKKS scheme (**??**), we will use V^{-1} to encode a complex vector into a real number vector, and V^T to decode a real number vector into a complex vector (**??**).

Condition for μ: It's worthwhile to note that the property $V \cdot V^T = n \cdot I_n^R$ does not hold if μ (denoting the μ-th cyclotomic polynomial) is not a power of 2. In particular, step 3 of the proof does not hold anymore if μ is not a power of 2:
$\sum_{k=0}^{n-1} \omega^{2k} \neq \sum_{k=0}^{n-1} \omega^{4k} \neq \sum_{k=0}^{n-1} \omega^{6k} \neq \cdots \neq \sum_{k=0}^{n-1} \omega^{2(n-1)k} \neq \sum_{k=0}^{n-1} \omega^{2(n+1)k} \neq \cdots \neq \sum_{k=0}^{n-1} \omega^{2(2n-1)k} \neq 0$

A-11.5 Vandermonde Matrix with Roots of Cyclotomic Polynomial over Rings ($\mathbb{Z}_p$)

Theorem **??** (in **??**) showed that $V \cdot V^T = n \cdot I_n^R$, where V is the Vandermonde matrix $V = Vander(x_0, x_1, \cdots, x_{n-1})$, where each x_i is the primitive μ-th root of unity over $X \in \mathbb{C}$ (i.e., complex number) and μ is a power of 2. In this subsection, we will show that the relation $V \cdot V^T = n \cdot I_n^R$ holds even if each x_i is the primitive μ-th root of unity over $X \in \mathbb{Z}_p$ (i.e., ring). In particular, we will prove Theorem **??**:

⟨Theorem ??⟩ Vandermonde Matrix with Roots of (power-of-2)-th Cyclotomic Polynomial over Ring ($\mathbb{Z}_p$)

The proof takes the same format as that of Theorem **??** (in **??**). Suppose we have an $n \times n$ (where n is a power of 2) Vandermonde matrix comprised of n distinct roots of the μ-th cyclotomic polynomial over $X \in \mathbb{Z}_p$ (ring), where μ is a power of 2 and $n = \frac{\mu}{2}$. In other words, $V = Vander(x_0, x_1, \cdots, x_{n-1})$, where each x_i is the root of $X^n + 1$ (i.e., the primitive ($\mu = 2n$)-th roots of unity). Then, the following holds:

$$V \cdot V^T = \begin{bmatrix} 0 & \cdots & 0 & 0 & n \\ 0 & \cdots & 0 & n & 0 \\ 0 & \cdots & n & 0 & 0 \\ \vdots & \iddots & \vdots & \vdots & \vdots \\ n & 0 & 0 & \cdots & 0 \end{bmatrix} = n \cdot I_n^R$$

And $V^{-1} = n^{-1} \cdot V^T \cdot I_n^R$

Proof.

1. $V \cdot V^T$ is expanded as follows:

$$V \cdot V^T = \begin{bmatrix} 1 & (\omega) & (\omega)^2 & \cdots & (\omega)^{n-1} \\ 1 & (\omega^3) & (\omega^3)^2 & \cdots & (\omega^3)^{n-1} \\ 1 & (\omega^5) & (\omega^5)^2 & \cdots & (\omega^5)^{n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & (\omega^{2n-1}) & (\omega^{2n-1})^2 & \cdots & (\omega^{2n-1})^{n-1} \end{bmatrix} \cdot \begin{bmatrix} 1 & 1 & 1 & \cdots & 1 \\ (\omega) & (\omega^3) & (\omega^5) & \cdots & (\omega^{2n-1}) \\ (\omega)^2 & (\omega^3)^2 & (\omega^5)^2 & \cdots & (\omega^{2n-1})^2 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ (\omega)^{n-1} & (\omega^3)^{n-1} & (\omega^5)^{n-1} & \cdots & (\omega^{2n-1})^{n-1} \end{bmatrix}$$

$$= \begin{bmatrix} \sum_{k=0}^{n-1} \omega^{2k} & \sum_{k=0}^{n-1} \omega^{4k} & \sum_{k=0}^{n-1} \omega^{6k} & \cdots & \sum_{k=0}^{n-1} \omega^{2nk} \\ \sum_{k=0}^{n-1} \omega^{4k} & \sum_{k=0}^{n-1} \omega^{6k} & \sum_{k=0}^{n-1} \omega^{8k} & \cdots & \sum_{k=0}^{n-1} \omega^{2k(n+1)} \\ \sum_{k=0}^{n-1} \omega^{6k} & \sum_{k=0}^{n-1} \omega^{8k} & \sum_{k=0}^{n-1} \omega^{10k} & \cdots & \sum_{k=0}^{n-1} \omega^{2k(n+2)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \sum_{k=0}^{n-1} \omega^{2nk} & \sum_{k=0}^{n-1} \omega^{2(n+1)k} & \sum_{k=0}^{n-1} \omega^{2(n+2)k} & \cdots & \sum_{k=0}^{n-1} \omega^{2(n+n-1)k} \end{bmatrix}$$

, where ω (i.e., the primitive ($\mu = 2n$)-th root of unity) has the order $2n$.

2. Note that the $V \cdot V^T$ matrix's anti-diagonal elements are $\sum_{k=0}^{n-1} \omega^{2nk}$. It can be seen that $\omega^{2n} \equiv 1 \bmod p$, because $\mathsf{Ord}_p(\omega) = 2n$. Thus, the $V \cdot V^T$ matrix's every anti-diagonal element is $\sum_{k=0}^{n-1} 1 = n$.

3. Next, we will prove that the $V \cdot V^T$ matrix has 0 for all other positions than the anti-diagonal ones. In other words, we will prove the following:

$$\sum_{k=0}^{n-1} \omega^{2k} = \sum_{k=0}^{n-1} \omega^{4k} = \sum_{k=0}^{n-1} \omega^{6k} = \cdots = \sum_{k=0}^{n-1} \omega^{2(n-1)k} = \sum_{k=0}^{n-1} \omega^{2(n+1)k} = \cdots = \sum_{k=0}^{n-1} \omega^{2(2n-1)k} = 0$$

The above is true, because in the particular case of the ($\mu = 2n$)-th cyclotomic polynomial $X^n + 1$ (where n is a power of 2), $\omega^i + \omega^{i+\frac{n}{2}} \equiv 0 \bmod p$ for each integer i where $0 \leq i \leq n-1$. Therefore, in the element $\sum_{k=0}^{n-1} \omega^{2k}$, its one-half terms add with its other-half terms and their final summation becomes 0. This is the same for all other following elements:
$\sum_{k=0}^{n-1} \omega^{4k}$, $\sum_{k=0}^{n-1} \omega^{6k}$, $\cdots$, $\sum_{k=0}^{n-1} \omega^{2(n-1)k}$, $\sum_{k=0}^{n-1} \omega^{2(n+1)k}$, $\cdots$, $\sum_{k=0}^{n-1} \omega^{2(2n-1)k}$

4. According to step 1 and 2, the $V \cdot V^T$ matrix has n on its anti-diagonal positions and 0 for all other positions.
5. Now we will derive the formula for V^{-1}. Given $V \cdot V^T = n \cdot I_n^R$,
$V^{-1} \cdot V \cdot V^T = V^{-1} \cdot n \cdot I_n^R$
$V^T = V^{-1} \cdot n \cdot I_n^R$
$V^T \cdot I_n^R = V^{-1} \cdot n \cdot I_n^R \cdot I_n^R$
$V^T \cdot I_n^R = V^{-1} \cdot n$ # since $I_n^R \cdot I_n^R = I_n$

Now, there is one caveat: modulo operation does not support direct number division (as explained in **??**). This means that the formula $V^{-1} = \frac{V^T \cdot I_n^R}{n}$ in Theorem **??** (in **??**) is inapplicable in our case, because our modulo p arithmetic does not allow direct division of $V^T \cdot I_n^R$ by n. Therefore, we instead multiply $V^T \cdot I_n^R$ by the inverse of n (i.e., n^{-1}). We continue as follows:
$V^T \cdot I_n^R = V^{-1} \cdot n$
$V^T \cdot I_n^R \cdot n^{-1} = V^{-1} \cdot n \cdot n^{-1}$
$V^{-1} = n^{-1} \cdot V^T \cdot I_n^R$

□

We finally proved that $V \cdot V^T = n \cdot I_n^R$, and $V^{-1} = n^{-1} \cdot V^T \cdot I_n^R$. Later in the BFV scheme (**??**), we will use V^{-1} to encode an integer vector into a vector of polynomial coefficients, and V^T to decode it back to the integer vector (**??**).

Condition for μ: Like in CKSS, it's worthwhile to note that the property $V \cdot V^T = n \cdot I_n^R$ does not hold if μ (denoting the μ-th cyclotomic polynomial) is not a power of 2. In particular, step 3 of the proof does not hold anymore if μ is not a power of 2:
$\sum_{k=0}^{n-1} \omega^{2k} \neq \sum_{k=0}^{n-1} \omega^{4k} \neq \sum_{k=0}^{n-1} \omega^{6k} \neq \cdots \neq \sum_{k=0}^{n-1} \omega^{2(n-1)k} \neq \sum_{k=0}^{n-1} \omega^{2(n+1)k} \neq \cdots \neq \sum_{k=0}^{n-1} \omega^{2(2n-1)k} \neq 0$

A-12 Modulo Rescaling

A-12.1 Rescaling Modulo of Congruence Relations

Remember from **??** that $a \bmod q$ is the remainder of a divided by k, and the congruence relation $a \equiv b \bmod q$ means that the remainder of a divided by q is the same as the remainder of b divided by q. Its equivalent numeric equation is $a = b + k \cdot q$, meaning that a and b differ by some multiple of q. The congruence and equation are two different ways of describing the relationship between two numbers a and b.

In this section, we introduce another way of describing the relationship between numbers. We will describe two numbers a and b in terms of a different modulo q' instead of the original modulo q. Such a change of modulo of a congruence relation is called modulo scaling. When we rescale the modulo of a congruence relation, we also need to rescale the numbers involved in the congruence relation.

Suppose we have the following congruence relations:
$a \equiv b \bmod q$
$a + c \equiv b + d \bmod q$
$a \cdot c \equiv b \cdot d \bmod q$

Now, suppose we want to rescale the modulo of the above congruence relations from $q \to q'$, where $q' \mid q$ (i.e., q' divides all of q, or q is a multiple of q'). Then, the accordingly updated congruence relations are as shown in **??**.

Congruence Relation	**Rescaled Congruence Relation – Exact**	**Rescaled Congruence Relation – Approximate**
$a \equiv b \bmod q$	$\left\lceil a\frac{q'}{q} \right\rfloor \equiv \left\lceil b\frac{q'}{q} \right\rfloor \bmod q'$ (if q divides all of: aq', bq')	$\left\lceil a\frac{q'}{q} \right\rfloor \cong \left\lceil b\frac{q'}{q} \right\rfloor \bmod q'$ (if q does not divide: aq' or bq')
$a + c \equiv b + d \bmod q$	$\left\lceil a\frac{q'}{q} \right\rfloor + \left\lceil c\frac{q'}{q} \right\rfloor \equiv \left\lceil b\frac{q'}{q} \right\rfloor + \left\lceil d\frac{q'}{q} \right\rfloor \bmod q'$ (if q divides all of: aq', bq', cq', dq')	$\left\lceil a\frac{q'}{q} \right\rfloor + \left\lceil c\frac{q'}{q} \right\rfloor \cong \left\lceil b\frac{q'}{q} \right\rfloor + \left\lceil d\frac{q'}{q} \right\rfloor \bmod q'$ (if q does not divide: aq', bq', cq', or dq')
$a \cdot c \equiv b \cdot d \bmod q$	$\left\lceil ac\frac{q'}{q} \right\rfloor \equiv \left\lceil bd\frac{q'}{q} \right\rfloor \bmod q'$ (if q divides both: acq', bdq')	$\left\lceil ac\frac{q'}{q} \right\rfloor \cong \left\lceil bd\frac{q'}{q} \right\rfloor \bmod q'$ (if q does not divide: acq' or bdq')

Table 3: Rescaling the congruence relations from modulo $q \to q'$ (where $\lceil \rfloor$ rounds to the nearest integer)

Proof.
1. $a \equiv b \bmod q \iff a = b + q \cdot k$ (for some integer k)
$\iff a \cdot \frac{q'}{q} = b \cdot \frac{q'}{q} + q \cdot k \cdot \frac{q'}{q}$
$\iff a \cdot \frac{q'}{q} = b \cdot \frac{q'}{q} + k \cdot q'$
   (a) If q divides all of aq', bq', cq', and dq', then $a \cdot \frac{q'}{q} = \left\lceil a\frac{q'}{q} \right\rfloor$, $b \cdot \frac{q'}{q} = \left\lceil b\frac{q'}{q} \right\rfloor$. Therefore:

$$a \cdot \frac{q'}{q} = b \cdot \frac{q'}{q} + k \cdot q'$$
$$\iff \left\lceil a\frac{q'}{q} \right\rceil = \left\lceil b\frac{q'}{q} \right\rceil + k \cdot q'$$
$$\iff \left\lceil a\frac{q'}{q} \right\rceil \equiv \left\lceil b\frac{q'}{q} \right\rceil \bmod q' \quad (\iff a \equiv b \bmod q)$$

(b) If q does not divide any of aq', bq', cq', or dq', then $a \cdot \frac{q'}{q} \approx \left\lceil a\frac{q'}{q} \right\rceil$, $b \cdot \frac{q'}{q} \approx \left\lceil b\frac{q'}{q} \right\rceil$. Therefore:

$$a \cdot \frac{q'}{q} = b \cdot \frac{q'}{q} + k \cdot q'$$
$$\iff \left\lceil a\frac{q'}{q} \right\rceil \approx \left\lceil b\frac{q'}{q} \right\rceil + k \cdot q'$$
$$\iff \left\lceil a\frac{q'}{q} \right\rceil \cong \left\lceil b\frac{q'}{q} \right\rceil \bmod q' \quad (\iff a \equiv b \bmod q)$$

2. $a + c \equiv b + d \bmod q \iff a + c = b + d + q \cdot k$ (for some integer q)
$$\iff a \cdot \frac{q'}{q} + c \cdot \frac{q'}{q} = b \cdot \frac{q'}{q} + d \cdot \frac{q'}{q} + q \cdot k \cdot \frac{q'}{q}$$
$$\iff a \cdot \frac{q'}{q} + c \cdot \frac{q'}{q} = b \cdot \frac{q'}{q} + d \cdot \frac{q'}{q} + k \cdot q'$$

(a) If q divides all of aq', bq', cq', and dq', then
$$a\frac{q'}{q} + c\frac{q'}{q} = \left\lceil a\frac{q'}{q} \right\rceil + \left\lceil c\frac{q'}{q} \right\rceil, \quad b\frac{q'}{q} + d\frac{q'}{q} = \left\lceil b\frac{q'}{q} \right\rceil + \left\lceil d\frac{q'}{q} \right\rceil$$
Therefore:
$$a \cdot \frac{q'}{q} + c \cdot \frac{q'}{q} = b \cdot \frac{q'}{q} + d \cdot \frac{q'}{q} + k \cdot q'$$
$$\iff \left\lceil a\frac{q'}{q} \right\rceil + \left\lceil c\frac{q'}{q} \right\rceil = \left\lceil b\frac{q'}{q} \right\rceil + \left\lceil d\frac{q'}{q} \right\rceil + k \cdot q'$$
$$\iff \left\lceil a\frac{q'}{q} \right\rceil + \left\lceil c\frac{q'}{q} \right\rceil \equiv \left\lceil b\frac{q'}{q} \right\rceil + \left\lceil d\frac{q'}{q} \right\rceil \bmod q' \quad (\iff a \equiv b \bmod q)$$

(b) If q does not divide any of aq', bq', cq', or dq', then
$$a\frac{q'}{q} + c\frac{q'}{q} \approx \left\lceil a\frac{q'}{q} \right\rceil + \left\lceil c\frac{q'}{q} \right\rceil, \quad b\frac{q'}{q} + d\frac{q'}{q} \approx \left\lceil b\frac{q'}{q} \right\rceil + \left\lceil d\frac{q'}{q} \right\rceil$$
Therefore:
$$a \cdot \frac{q'}{q} + c \cdot \frac{q'}{q} = b \cdot \frac{q'}{q} + d \cdot \frac{q'}{q} + k \cdot q'$$
$$\iff \left\lceil a\frac{q'}{q} \right\rceil + \left\lceil c\frac{q'}{q} \right\rceil \approx \left\lceil b\frac{q'}{q} \right\rceil + \left\lceil d\frac{q'}{q} \right\rceil + k \cdot q'$$
$$\iff \left\lceil a\frac{q'}{q} \right\rceil + \left\lceil c\frac{q'}{q} \right\rceil \cong \left\lceil b\frac{q'}{q} \right\rceil + \left\lceil d\frac{q'}{q} \right\rceil \bmod q' \quad (\iff a + c \equiv b + d \bmod q)$$

3. $a \cdot c \equiv b \cdot d \bmod q \iff a \cdot c = b \cdot d + q \cdot k$ (for some integer q)

$\iff ac \cdot \frac{q'}{q} = bd \cdot \frac{q'}{q} + q \cdot k \cdot \frac{q'}{q}$

$\iff ac \cdot \frac{q'}{q} = bd \cdot \frac{q'}{q} + k \cdot q'$

(a) If q divides all of aq', bq', cq', and dq', then

$ac \cdot \frac{q'}{q} = \left\lceil ac\frac{q'}{q} \right\rfloor, \quad bd \cdot \frac{q'}{q} = \left\lceil bd\frac{q'}{q} \right\rfloor$

Therefore:

$ac \cdot \frac{q'}{q} = bd \cdot \frac{q'}{q} + k \cdot q'$

$\iff \left\lceil ac\frac{q'}{q} \right\rfloor = \left\lceil bd\frac{q'}{q} \right\rfloor + k \cdot q'$

$\iff \left\lceil ac\frac{q'}{q} \right\rfloor \equiv \left\lceil bd\frac{q'}{q} \right\rfloor \bmod q' \quad (\iff a \cdot c \equiv b \cdot d \bmod q)$

(b) If q does not divide any of aq', bq', cq', or dq', then

$ac \cdot \frac{q'}{q} \approx \left\lceil ac\frac{q'}{q} \right\rfloor, \quad bd \cdot \frac{q'}{q} \approx \left\lceil bd\frac{q'}{q} \right\rfloor$

Therefore:

$ac \cdot \frac{q'}{q} = bd \cdot \frac{q'}{q} + k \cdot q'$

$\iff \left\lceil ac\frac{q'}{q} \right\rfloor \approx \left\lceil bd\frac{q'}{q} \right\rfloor + k \cdot q'$

$\iff \left\lceil ac\frac{q'}{q} \right\rfloor \cong \left\lceil bd\frac{q'}{q} \right\rfloor \bmod q' \quad (\iff a \cdot c \equiv b \cdot d \bmod q)$

□

As shown in the proof, if all numbers in the congruence relations are exactly divisible by the rescaling factor during the modulo rescaling, then the rescaled result gives exact congruence relations in the new modulo. On the other hand, if any numbers in the congruence relations are not divisible by the rescaling factor during the modulo rescaling (i.e., we need to round some decimals), then the rescaled result gives approximate congruence relations in the new modulo.

In a more complicated congruence relation that contains many $(+, -, \cdot)$ operations, the same principle of modulo rescaling explained above can be recursively applied to each pair of operands surrounding each operator.

A-12.1.1 Example

Suppose we have the following congruence relation:

$b \equiv a \cdot s + \Delta \cdot m + e \bmod q, \quad$ where: $q = 30, \quad s = 5, \quad a = 10, \quad \Delta = 10, \quad m = 1, \quad e = 10, \quad b = 40$

First, we can test if the above congruence relation is true by plugging in the given example values as follows:

$b \equiv a \cdot s + \Delta \cdot m + e \bmod 30$

$40 \equiv 10 \cdot 5 + 10 \cdot 1 + 10 \bmod 30$
$40 \equiv 70 \bmod 30$

This congruence relation is true.

Now, suppose we want to rescale the modulo from $30 \to 3$. Then, based on the rescaling principles described in **??**, we compute the rescaled values as follows:

$q' = 3, \quad s = 5, \quad m = 1$

$$\hat{a} = \left\lceil a \cdot \frac{3}{30} \right\rceil = \left\lceil 10 \cdot \frac{3}{30} \right\rceil = 1$$

$$\hat{\Delta} = \left\lceil \Delta \cdot \frac{3}{30} \right\rceil = \left\lceil 10 \cdot \frac{3}{30} \right\rceil = 1$$

$$\hat{e} = \left\lceil e \cdot \frac{3}{30} \right\rceil = \left\lceil 10 \cdot \frac{3}{30} \right\rceil = 1$$

$$\hat{b} = \left\lceil b \cdot \frac{3}{30} \right\rceil = \left\lceil 40 \cdot \frac{3}{30} \right\rceil = 4$$

The rescaled congruence relation from modulo $30 \to 3$ is derived as follows:

$$\left\lceil b\frac{3}{30} \right\rceil \equiv \left\lceil s \cdot a\frac{3}{30} \right\rceil + \left\lceil m \cdot \Delta\frac{3}{30} \right\rceil + \left\lceil e\frac{3}{30} \right\rceil \bmod 3$$

$\hat{b} \equiv \hat{a} \cdot s + \hat{\Delta} \cdot m + \hat{e} \bmod 3$ (an exact congruence relation, as all rescaled values have no decimals)

$4 \equiv 1 \cdot 5 + 1 \cdot 1 + 1 \bmod 3$

$4 \equiv 7 \bmod 3$

As shown above, the rescaled congruence relation preserves correctness, because all rescaled values are divisible by the rescaling factor. By contrast, if $\frac{q}{q'} = \frac{30}{3} = 10$ did not divide any of $a \cdot s$, Δm, or e, then the rescaled congruence relation would be an approximate (i.e., $\cong$) congruence relation.

A-13 Chinese Remainder Theorem

- **Reference 1:** Brilliant – Chinese Remainder Theorem [**?**]
- **Reference 2:** YouTube – Extended Euclidean Algorithm Tutorial

⟨Theorem ??.1⟩ Chinese Remainder Theorem

Suppose we have positive coprime integers $n_0, n_1, n_2, \cdots, n_k$. Let $N = n_1 n_2 \cdots n_k$. We sample $k+1$ random integers $a_0, a_1, a_2, \cdots, a_k$ from each modulo domain $n_0, n_1, n_2, \ldots, n_k$ (i.e., $a_0 \in \mathbb{Z}_{n_0}$, $a_1 \in \mathbb{Z}_{n_1}$, $\cdots$, $a_k \in \mathbb{Z}_{n_k}$). Then, there exists one and only one solution $x \bmod N$ such that x is congruent with $a_0, a_1, a_2, \cdots, a_k$ in each modulo $n_0, n_1, n_2, \cdots, n_k$. That is:

$$\begin{aligned} x &\equiv a_0 \bmod n_0 \\ x &\equiv a_1 \bmod n_1 \\ x &\equiv a_2 \bmod n_2 \\ &\vdots \\ x &\equiv a_k \bmod n_k \end{aligned}$$

To compute x, we first compute each y_i and z_i (for $0 \le i \le k$) as follows:

$$y_i = \frac{N}{n_i}, \quad z_i = y_i^{-1} \bmod n_i$$

Note that each y_i's inverse (i.e., y_i^{-1}) can be computed by using the Extended Euclidean algorithm (watch the YouTube tutorial). Then, the unique solution x can be computed as follows:

$$x = \sum_{i=0}^{k} a_i y_i z_i \ \# \text{ Alternatively, we can compute } x = \sum_{i=0}^{k} |a_i z_i|_{n_i} y_i \text{ (where } |a_i z_i|_{n_i} = a_i z_i \bmod n_i)$$

Since such x is unique in mod N, there are isomorphic mappings between $x \bmod N$ and $(a_0, a_1, a_2, \cdots, a_k)$.

Also, $y_i z_i \equiv (y_i z_i)^2 \bmod N \quad$ for all $0 \le i \le k$

Proof.

1. Given $x = \sum_{i=0}^{k} a_i y_i z_i$, let's compute $x \bmod n_i$ for each i where $0 \le i \le k$:

$$\begin{aligned} x \bmod n_i &= \sum_{i=0}^{k} a_i y_i z_i \bmod n_i \\ &= a_0 y_0 z_0 + a_1 y_1 z_1 + a_2 y_2 z_2 + \cdots + a_k y_k z_k \bmod n_i \\ &= a_i y_i z_i \bmod n_i \ \# \text{ because } y_j \equiv 0 \bmod n_i \text{ for all } j \neq i, \text{ as they are a multiple of } n_i \\ &= a_i \ \# \text{ because } y_i z_i = y_i y_i^{-1} = 1 \end{aligned}$$

Thus, the value of x in each modulo $n_0, n_1, n_2, \cdots, n_k$ is congruent with $a_0, a_1, a_2, \cdots, a_k$.

Alternatively, note that the following is also true:

$x \bmod n_i = \sum_{i=0}^{k} |a_i z_i|_{n_i} y_i \pmod{n_i}$
$= |a_0 z_0|_{n_0} y_0 + |a_1 z_1|_{n_1} y_1 + |a_2 z_2|_{n_2} y_2 + \cdots + |a_k y_k|_{n_k} y_k \bmod n_i$
$= |a_i z_i|_{n_i} y_i \bmod n_i$
$= a_i$

2. Now, we prove that x is a unique solution in modulo N. Suppose there were two solutions: x and x' such that:
$x \equiv x' \equiv a_0 \bmod n_0$
$x \equiv x' \equiv a_1 \bmod n_1$
$x \equiv x' \equiv a_2 \bmod n_2$
$\vdots$
$x \equiv x' \equiv a_k \bmod n_k$

   Then, by definition of modulo congruence, $n_0 \mid (x-x'), n_1 \mid (x-x'),\ n_2 \mid (x-x'),\ \cdots,\ n_k \mid (x-x')$.
   Also, since $n_0, n_1, n_2, \cdots, n_k$ are coprime, it must be the case that $n_0 n_1 n_2 n_3 \cdots n_k \mid (x - x')$, or $N \mid (x - x')$. This means that $x \equiv x' \bmod N$. Therefore, x is a unique solution in modulo N.

3. Now, we will prove that $y_i z_i \equiv (y_i z_i)^2 \bmod N \quad$ for all $0 \leq i \leq k$.

   In the case of modulo n_i, $y_i z_i \equiv 1 \bmod n_i$, since z_i is an inverse of y_i modulo n_j. In the case of all other modulo n_j where $i \neq j$, $y_i z_i \equiv 0 \bmod n_j$, because $y_i = \dfrac{N}{n_i}$ and thus n_j divides y_i.

   By squaring both sides of $(y_i z_i) \equiv 1 \bmod n_i$, we get $(y_i z_i)^2 \equiv 1 \bmod n_i$. Similarly, by squaring both sides of $(y_i z_i) \equiv 1 \bmod n_j$, we get $(y_i z_i)^2 \equiv 0 \bmod n_j$.

   Therefore, $y_i z_i - (y_i z_i)^2 \equiv 0 \bmod n_i$, and $y_i z_i - (y_i z_i)^2 \equiv 0 \bmod n_j$. In other words, $y_i z_i - (y_i z_i)^2 \equiv 0 \bmod n_j$ for all $0 \leq j \leq k$.

   Then, we do the similar reasoning as step 2: since every co-prime n_j divides $y_i z_i - (y_i z_i)^2$, $n_0 n_1 \cdots n_k = N$ divides $y_i z_i - (y_i z_i)^2$. Thus, $y_i z_i - (y_i z_i)^2 \equiv 0 \bmod N$, which is $y_i z_i \equiv (y_i z_i)^2 \bmod N$. This is true for all $0 \leq i \leq k$.

□

A-13.1 Application: Residue Number System (RNS)

In a modern processor, each data size is a maximum of 64 bits. If the data size exceeds 64 bits, its computations can be handled efficiently by using the Chinese remainder theorem such that each co-prime modulus $\log_2 n_i \leq 64$ (where $N = n_0 \cdot n_1 \cdots \cdot n_k$) can be used to represent a big value $a \bmod N$ as $\vec{a}_{crt} = (a_0, a_1, \cdots, a_k)$, where $a \equiv a_i \bmod n_i$. Then, for any pair of big numbers a and $b \bmod N$, we can compute $a + b \bmod N$ and $a \cdot b \bmod N$ as follows:

- $a + b \equiv \sum_{i=0}^{k} a_i y_i z_i + \sum_{i=0}^{k} b_i y_i z_i \equiv \sum_{i=0}^{k} a_i y_i z_i + b_i y_i z_i \equiv \sum_{i=0}^{k} (a_i + b_i) y_i z_i \bmod N$

- $a \cdot b \equiv \sum_{i=0}^{k} a_i y_i z_i \cdot \sum_{i=0}^{k} b_i y_i z_i \equiv \sum_{i=0}^{k} (a_i \cdot b_i)(y_i z_i)^2 + \sum_{i \neq j}^{k} (a_i \cdot b_j) y_i z_i y_j z_j \equiv \sum_{i=0}^{k} (a_i \cdot b_i)(y_i z_i)^2$
  # Note that all terms $y_i z_i y_j z_j$ where $i \neq j$ are 0 modulo N, because $y_i y_j \bmod N \equiv 0$.
  This is because $y_i = n_0 n_1 \cdots n_{i-1} n_{i+1} \cdots$ and $y_j = n_0 n_1 \cdots n_{j-1} n_{j+1} \cdots$.
  Thus $y_i y_j$ is a multiple of N.

  $\equiv \sum_{i=0}^{k} (a_i \cdot b_i)(y_i z_i) \bmod N$
  # This is because $(y_i z_i) \equiv (y_i z_i)^2$ as shown in step 3 in the proof of Theorem **??**.1

Thus, the Chinese remainder theorem gives us the following useful formula:

⟨Theorem ??.2⟩ Application of the Chinese Remainder Theorem

Suppose there are two big numbers $a = \sum_{i=0}^{k} a_i y_i z_i \bmod N$ and $b = \sum_{i=0}^{k} b_i y_i z_i \bmod N$ where N is a multiplication of co-primes $n_1 n_2 \cdots n_k$, we have an isomorphism as follows:
$a \xrightarrow{\sigma} \vec{a}_{crt} = (a_0, a_1, \cdots, a_k)$
$b \xrightarrow{\sigma} \vec{b}_{crt} = (b_0, b_1, \cdots, b_k)$

Based on the above isomorphism, the following is true:

- $a + b \equiv \sum_{i=0}^{k} (a_i + b_i) y_i z_i \bmod N \iff \vec{a}_{crt} + \vec{b}_{crt} \equiv (a_0 + b_0,\ a_1 + b_1, \cdots,\ a_k + b_k) \bmod N$
- $a \cdot b \equiv \sum_{i=0}^{k} (a_i \cdot b_i) y_i z_i \bmod N \iff \vec{a}_{crt} \odot \vec{b}_{crt} \equiv (a_0 b_0, a_1 b_1, \cdots, a_k b_k) \bmod N$

, where each element-wise addition/multiplication can be independently done modulo n_i

A-14 Taylor Series

The Taylor series is a mathematical formula to approximate a complex equation as a polynomial. Formally speaking, the Taylor series of a function is an infinite sum of the evaluation of the function's derivatives at a single point. Given function $f(X)$, its Taylor series evaluated at $X = a$ is expressed as follows:

$$f(a) + \frac{f'(a)}{1!}(X-a) + \frac{f''(a)}{2!}(X-a)^2 + \frac{f'''(a)}{3!}(X-a)^3 + \cdots = \sum_{d=0}^{\infty} \frac{f^{(d)}(a)}{d!}(X-a)^d$$

For a target function, the Taylor series can aggregate a finite number of terms, D, instead of ∞ terms. Such a D-degree polynomial is also called the D-th Taylor polynomial approximating $f(X)$. The higher the total number of degrees D is, the more accurate the approximation of $f(X)$ becomes. The accuracy of the approximation is higher for those coordinates nearby $X = a$, and lower for those coordinates away from $X = a$. To increase the accuracy for further-away coordinates, we need to increase D.

A-15 Lagrange's Polynomial Interpolation

Suppose we are given $n+1$ two-dimensional coordinates $(x_0, y_0), (x_1, y_1), \cdots, (x_n, y_n)$, whereas all x values are distinct but y values are not necessarily distinct. Lagrange's polynomial interpolation is a technique to find a unique n-degree polynomial that passes through such $n+1$ coordinates. The domain of X and Y values is complex numbers (which include the real domain), or can be modulo prime.

⟨Theorem ??⟩ Lagrange's Polynomial Interpolation

Suppose we are given $n+1$ two-dimensional coordinates $(x_0, y_0), (x_1, y_1), \cdots, (x_n, y_n)$, whereas all X values are distinct but the Y values don't need to be distinct. The domain of (X, Y) can be either: $(x_i, y_i) \in \mathbb{C}^2$ (which includes the real domain) or $(x_i, y_i) \in \mathbb{Z}_p^2$ (where p is a prime). Then, there exists a unique n-degree (or lesser degree) polynomial $f(X)$ that passes through these n coordinates. Such a polynomial $f(X)$ is computed as follows:

$$f(X) = \sum_{j=0}^{n} \frac{(X-x_0)\cdot(X-x_1)\cdots(X-x_{j-1})\cdot(X-x_{j+1})\cdots(X-x_n)}{(x_j-x_0)\cdot(x_j-x_1)\cdots(x_j-x_{j-1})\cdot(x_j-x_{j+1})\cdots(x_j-x_n)} \cdot y_j$$

$$f(X) = \sum_{j=0}^{n} \left(\prod_{\substack{0 \le k \le n \\ k \ne j}} \frac{X-x_k}{x_j-x_k} \cdot y_j \right)$$

Proof.

1. First, we will show that there exists an n-degree (or lesser degree) polynomial $f(X)$ that passes through the $n+1$ distinct coordinates: $(x_0, y_0), (x_1, y_1), \ldots, (x_n, y_n)$. Such a polynomial $f(X)$ is designed as follows:

$$\begin{aligned} f(X) &= \sum_{j=0}^{n} \frac{(X-x_0)\cdot(X-x_1)\cdots(X-x_{j-1})\cdot(X-x_{j+1})\cdots(X-x_n)}{(x_j-x_0)\cdot(x_j-x_1)\cdots(x_j-x_{j-1})\cdot(x_j-x_{j+1})\cdots(x_j-x_n)} \cdot y_j \\ &= \sum_{j=0}^{n} \left(\prod_{\substack{0 \le k \le n \\ k \ne j}} \frac{X-x_k}{x_j-x_k} \cdot y_j \right) \\ &= \sum_{j=0}^{n} \ell_j(X) \cdot y_j \ \# \text{ where } \ell_j(X) = \sum_{j=0}^{n} \left(\prod_{\substack{0 \le k \le n \\ k \ne j}} \frac{X-x_k}{x_j-x_k} \right) \end{aligned}$$

   We call $\{\ell_0(X), \ell_1(X), \ldots, \ell_n(X)\}$ the Lagrange basis for polynomials of degree $\le n$. Given this design of $f(X)$, notice that for each of $(x_i, y_i) \in \{(x_0, y_0), (x_1, y_1), \ldots, (x_n, y_n)\}$, $\ell_i(x_{i'}) = 1$ for $i' = i$, and $\ell_i(x_{i'}) = 0$ for $i' \ne i$. Therefore, $f(x_i) = \sum_{j=0}^{n} \ell_j(x_i) \cdot y_j = 1 \cdot y_i = y_i$ for $0 \le i \le n$. In other words, $f(X)$ passes through the $n+1$ distinct coordinates: $\{(x_0, y_0), (x_1, y_1), \ldots, (x_n, y_n)\}$. Such a satisfactory $f(X)$ can be computed in the case where the domain of (X, Y) is either: $(x_i, y_i) \in \mathbb{C}^2$ (i.e., real and complex numbers), or $(x_i, y_i) \in \mathbb{Z}_p^2$ (where p is a prime). Especially, a valid $f(X)$ can be computed also in the mod p domain, because as we learned from Fermat's Little Theorem in Theorem **??**.4 (**??**), $a^{p-1} \equiv 1 \bmod p$ if and only if a and p are co-prime, and this means that if p is a prime, then $a^{p-1} \equiv 1 \bmod p$ for all $a \in \mathbb{Z}_p^\times$ (i.e., $\mathbb{Z}_p$ without $\{0\}$). Since every value in $\mathbb{Z}_p^\times$ has an inverse, we can compute the denominator's division operation

in $f(X) = \sum_{j=0}^{n} \frac{(X-x_0)\cdot(X-x_1)\cdots(X-x_{j-1})\cdot(X-x_{j+1})\cdots(X-x_n)}{(x_j-x_0)\cdot(x_j-x_1)\cdots(x_j-x_{j-1})\cdot(x_j-x_{j+1})\cdots(x_j-x_n)} \cdot y_j$ by converting them into multiplication of their counter-part inverses, and thereby compute a validly defined $f(X) \bmod p$.

2. Next, we will prove that no two distinct n-degree (or lesser degree) polynomials $f_1(X)$ and $f_2(X)$ can pass through the same $n+1$ distinct (X, Y) coordinates. Suppose there exist such two polynomials $f_1(X)$ and $f_2(X)$. Then $f_{1-2}(X) = f_1(X) - f_2(X)$ will be a new n-degree (or lesser degree) polynomial that passes through $(x_0, 0), (x_1, 0), \cdots, (x_n, 0)$. This means that $f_{1-2}(X)$ has $n+1$ distinct roots. In other words, $f_{1-2}(X)$ is an $n+1$-degree (or higher-degree) polynomial. However, this contradicts the axiom that $f_{1-2}(X)$ is an n-degree (or lesser degree) polynomial. Therefore, there exist no two polynomials $f_1(X)$ and $f_2(X)$ that pass through the same $n+1$ distinct (X, Y) coordinates.
3. We have shown that there exists some n-degree (or lesser degree) polynomial $f(X)$ that passes through $n+1$ distinct (X, Y) coordinates, and no such two or more distinct polynomials exists. Therefore, there exists only a unique polynomial that satisfies this requirement.

□

A-16 Efficient Polynomial Multiplication by FFT and NTT

- Reference: Polynomials and the Fast Fourier Transform (FFT) [**?**]

A-16.1 Background and Motivation

Given two $(n-1)$-degree polynomials:

$A(X) = \sum_{i=0}^{n-1} a_i X^i, \quad B(X) = \sum_{i=0}^{n-1} b_i X^i$

, the polynomial multiplication $C(X) = A(X) \cdot B(X)$ is computed as follows:

$C(X) = \sum_{i=0}^{2n-1} c_i X^i$, where $c_i = \sum_{k=0}^{i} a_k b_{i-k}$

This operation of computing $\vec{c} = (c_0, c_1, \cdots, c_{2n-1})$ is also called the convolution of $\vec{a}$ and $\vec{b}$, denoted as $\vec{c} = \vec{a} \otimes \vec{b}$. The time complexity of this operation (i.e., the total number of multiplications between two numbers) is $O(n^2)$.

Another way of multiplying two polynomials is based on **point-value representation**. The point-value representation of an $(n-1)$-degree (or lesser degree) polynomial $A(X)$ is a set of n coordinates $\{(x_0, y_0), (x_1, y_1), \cdots (x_{n-1}, y_{n-1})\}$, where each x_i is a distinct X coordinate (whereas each y_i is not necessarily a distinct Y coordinate). Given a point-value representation of an $(n-1)$-degree (or lesser degree) polynomial, we can use polynomial interpolation (**??**) to derive the polynomial. Let's denote the point-value representation of $(n-1)$-degree (or lesser degree) polynomial $A(X)$ and $B(X)$ as follows:

$A(X)$: $\big((x_0, y_0^{\langle a \rangle}), (x_1, y_1^{\langle a \rangle}), \cdots (x_{n-1}, y_{n-1}^{\langle a \rangle})\big)$

$B(X)$: $\big((x_0, y_0^{\langle b \rangle}), (x_1, y_1^{\langle b \rangle}), \cdots (x_{n-1}, y_{n-1}^{\langle b \rangle})\big)$

Then, the point-value representation of the polynomial $C(X) = A(X) \cdot B(X)$ can be computed as a Hadamard product (Definition **??** in **??**) of the y values of the point-value representation of $A(X)$ and $B(X)$ as follows:

$C(X)$: $\big((x_0, y_0^{\langle c \rangle}), (x_1, y_1^{\langle c \rangle}), \cdots (x_{n-1}, y_{n-1}^{\langle c \rangle})\big)$, where $y_i^{\langle c \rangle} = y_i^{\langle a \rangle} \cdot y_i^{\langle b \rangle}$

However, we cannot derive polynomial $C(X)$ based on these n coordinates because the degree of $C(X)$ is $2n-2$ (or less than $2n-2$). But if we regard all polynomials (including $A(X), B(X)$ and $C(X)$) to be in the polynomial ring $\mathbb{R}[X]/X^n + 1$ (or $\mathbb{Z}[X]_p/X^n + 1$), then we can reduce the $(2n-2)$-degree polynomial $C(X)$ to a congruent $(n-1)$-degree (or lesser degree) polynomial in the ring. Then, the $n-1$ coordinates of $C(X)$ are sufficient to derive $C(X)$.

However, the time complexity of this new method is still $O(n^2)$. The Hadamard product between two polynomials' point-value representations takes $O(n)$, but evaluating a polynomial at n distinct x values takes $O(n^2)$ (because each polynomial has n terms, and we have to compute each term for n distinct x values). The polynomial interpolation for deriving $C(X)$ also takes $O(n^2)$.

To solve this efficiency problem, this section will introduce an efficient technique for polynomial evaluation, which can evaluate a polynomial at n distinct roots of unity in $O(n \log n)$. This technique is classified into 2 types: Fast Fourier Transform (FFT) and Number-theoretic Transform(NTT). These two types are technically almost the same, with the only difference being that the FFT assumes a polynomial ring over complex numbers (**??**), whereas the NTT assumes a polynomial ring over the integers (**??**). Polynomial multiplication based on FFT (or NTT) comprises 3

steps: (1) forward FFT (or NTT); (2) point-value multiplication; and (3) inverse FFT (or NTT).

A-16.2 Forward FFT (or NTT)

We assume a polynomial ring of $\mathbb{R}[X]/X^n+1$ for FTT, and $\mathbb{Z}[X]_p/X^n+1$ for NTT (where X^n+1 is a cyclotomic polynomial). The x coordinates to evaluate the target polynomial are the n distinct roots of X^n+1, which are $\omega^1, \omega^3, \ldots, \omega^{2n-1}$, where ω is the primitive $2n$-th root of unity. Then, the point-value representation of the polynomial $A(X)$ is $((x_0, y_0^{\langle a \rangle}), (x_1, y_1^{\langle a \rangle}), \cdots (x_{n-1}, y_{n-1}^{\langle a \rangle}))$, where:

$y_i^{\langle a \rangle} = A(\omega^{2i+1}) = \sum_{j=0}^{n-1} a_j \cdot (\omega^{2i+1})^j = \sum_{j=0}^{n-1} a_j \cdot \omega^{(2i+1)\cdot j}$.

We call the vector $\vec{y}^{\langle a \rangle} = (y_0^{\langle a \rangle}, y_1^{\langle a \rangle}, \cdots, y_{n-1}^{\langle a \rangle})$ the Discrete Fourier Transform (DFT) of the coefficient vector $\vec{a} = (a_0, a_1, \cdots, a_{n-1})$. We write this as $\vec{y}^{\langle a \rangle} = \mathsf{DFT}(\vec{a})$. As explained in **??**, the computation of the DFT takes $O(n^2)$, because we have to evaluate n distinct X values for a polynomial that has n terms.

A-16.2.1 High-level Idea

FFT (or NTT) is an improved method for computing the DFT, which reduces the time complexity from $O(n^2)$ to $O(n \log n)$. The high-level idea of FFT is to split the $(n-1)$-degree (or lesser degree) target polynomial $A(X)$ to evaluate into 2 half-degree polynomials $A_0(X)$ and $A_1(X)$ as follows:

$A(X) = a_0 + a_1X + a_2X^2 + \cdots + a_{n-1}X^{n-1}$
$\quad = A_0(X^2) + X \cdot A_1(X^2)$ $A_0(X) = a_0 + a_2X + a_4X^2 + \cdots + a_{n-2}X^{\frac{n}{2}-1}$
$A_1(X) = a_1 + a_3X + a_5X^2 + \cdots + a_{n-2}X^{\frac{n}{1}-1}$

The above method of splitting a polynomial into two half-degree polynomials is called the Cooley-Tukey step. As we split $A(X)$ into two smaller-degree polynomials $A_0(X)$ and $A_1(X)$, evaluating $A(X)$ at the odd-powered primitive $2n$-th roots of unity $\{\omega^1, \omega^3, \omega^5, \cdots, \omega^{2n-1}\}$ is equivalent to evaluating $A_0(X)$ and $A_1(X)$ at n distinct *squared* n-th roots of unity $\{(\omega^2)^1, (\omega^2)^3, (\omega^2)^5, \ldots, (\omega^2)^{2n-1}\}$ and computing $A_0(X^2) + X \cdot A_1(X^2)$. However, remember that the primitive $2n$-th root of unity ω has order $2n$ (i.e., $\omega^{2n} = 1$ and $\omega^m \neq 1$ for all $m < 2n$). Therefore, the second half of $\{(\omega^2)^1, (\omega^2)^3, (\omega^2)^5, \ldots, (\omega^2)^{2n-1}\}$ is a repetition of the first half. This implies that we only need to evaluate $A_0(X)$ and $A_1(X)$ at $\frac{n}{2}$ distinct x coordinates each, instead of n distinct coordinates, because the polynomial evaluation results for the other half are the same as those of the first half (as their input x to the polynomial is the same).

We recursively split $A_0(X)$ and $A_1(X)$ into half-degree polynomials and evaluate them at half-counted n-th roots of unity. Then, the total number of rounds of splitting is $\log n$, and the maximum number of root-to-coefficient multiplications in each round is n, which aggregates to $O(n \log n)$.

A-16.2.2 Details

Suppose we have a polynomial ring that is either $\mathbb{Z}_p[X]/X^8+1$ (i.e., over a finite field with prime p) or $\mathbb{R}[X]/X^8+1$ (over complex numbers). We denote the primitive $(2n=8)$-th roots of unity as ω, and the n distinct $(n=8)$-th roots of X^n+1 are: $\{\omega^1, \omega^3, \omega^5, \omega^7, \omega^9, \omega^1 1, \omega^1 3, \omega^1 5\}$.

Now, we define our target polynomial to evaluate as follows:
$A(X) = a_0 + a_1X + a_2X^2 + a_3X^3 + a_4X^4 + a_5X^5 + a_6X^6 + a_7X^7$

We split this 7-degree polynomial into the following two 3-degree polynomials (using the Cooley-Tukey step):

$A_0(X) = a_0 + a_2X + a_4X^2 + a_6X^3$
$A_1(X) = a_1 + a_3X + a_5X^2 + a_7X^3$
$A(X) = A_0(X^2) + X \cdot A_1(X^2)$

We recursively split the two 3-degree polynomials above into 1-degree polynomials as follows:

$A_{0,0}(X) = a_0 + a_4X, \quad A_{0,1}(X) = a_2 + a_6X$
$A_0(X) = A_{0,0}(X^2) + X \cdot A_{0,1}(X^2)$

$A_{1,0}(X) = a_1 + a_5X, \quad A_{1,1}(X) = a_3 + a_7X$
$A_1(X) = A_{1,0}(X^2) + X \cdot A_{1,1}(X^2)$

$$\begin{aligned} A(X) &= A_0(X^2) + X \cdot A_1(X^2) \\ &= \underbrace{\underbrace{(\underbrace{A_{0,0}(X^4)}_{\text{FFT Level 1}} + X^2 \cdot \underbrace{A_{0,1}(X^4)}_{\text{FFT Level 1}})}_{\text{FFT Level 2}} + X \cdot \underbrace{(\underbrace{A_{1,0}(X^4)}_{\text{FFT Level 1}} + X^2 \cdot \underbrace{A_{1,1}(X^4)}_{\text{FFT Level 1}})}_{\text{FFT Level 2}}}_{\text{FFT Level 3}} \end{aligned}$$

To evaluate $A(X)$ at the n distinct roots $X = \{\omega^1, \omega^3, \ldots, \omega^15\}$, we evaluate each FFT level of the above formula at $X = \{\omega^1, \omega^3, \cdots, \omega^15\}$, starting from level $1 \leq l \leq 3$.

FFT Level $l = 1$: We evaluate $A_{0,0}(X^4)$, $A_{0,1}(X^4)$, $A_{1,0}(X^4)$, and $A_{1,1}(X^4)$ at $X = \{\omega^1, \omega^3, \cdots, \omega^15\}$. However, notice that plugging in $X = \{\omega^1, \omega^3, \cdots, \omega^15\}$ to X^4 results in only 2 distinct values: ω^0 and ω^8. This is because the order of ω is $2n$ (i.e., $\omega^{2n} = 1$), and thus $(\omega^1)^4 = (\omega^5)^4 = (\omega^9)^4 = (\omega^{13})^4$, and $(\omega^3)^4 = (\omega^7)^4 = (\omega^{11})^4 = (\omega^{15})^4$. Therefore, we only need to evaluate $A_{0,0}(X^4)$, $A_{0,1}(X^4)$, $A_{1,0}(X^4)$, and $A_{1,1}(X^4)$ at 2 distinct x values instead of 8, where each evaluation requires a constant number of arithmetic operations: computing 1 multiplication and 1 addition. As there are a total of 4 polynomials to evaluate (i.e., $A_{0,1}(X^4), A_{0,1}(X^4), A_{1,0}(X^4), A_{1,1}(X^4)$), we compute the FFT a total of $4 \cdot 2 = 8$ times.

FFT Level $l = 2$: Based on the evaluation results from FFT Level 1 as building blocks, we evaluate $A_0(X^2)$ and $A_1(X^2)$ at $X = \{\omega^1, \omega^3, \cdots, \omega^{15}\}$. However, notice that plugging in $X = \{\omega^1, \omega^3, \cdots, \omega^15\}$ to X^2 results in only 4 distinct values: ω^1, ω^5, ω^9, and ω^{13}. This is because the order of ω is n (i.e., $\omega^n = 1$), and thus $(\omega^1)^2 = (\omega^9)^2$, $(\omega^3)^2 = (\omega^{11})^2$, $(\omega^5)^2 = (\omega^{13})^2$, and $(\omega^7)^2 = (\omega^{15})^2$. Therefore, we only need to evaluate $A_0(X^2)$ and $A_1(X^2)$ at 4 distinct x values instead of 8, where each evaluation requires a constant number of arithmetic operations: computing 1 multiplication and 1 addition (where we use the results from FFT Level 1 as building blocks, and the computational structure of FFT Level 2 is the same as that of FFT Level 1). There are a total of 2 polynomials to evaluate (i.e., $A_0(X^2), A_1(X^2)$); thus, we compute the FFT a total of $2 \cdot 4 = 8$ times.

FFT Level $l = 3$: Based on the evaluation results from FFT Level 2 as building blocks, we evaluate $A(X)$ at $X = \{\omega^1, \omega^3, \cdots, \omega^{15}\}$. For this last level of computation, we need to evaluate

all 8 distinct X values, since they are all unique values, and each evaluation requires a constant number of arithmetic operations: computing 1 multiplication and 1 addition. There is a total of 1 polynomial to evaluate (i.e., $A(X)$); thus, we compute the FFT a total of $1 \cdot 8 = 8$ times.

Generalization: Suppose that the degree of the target polynomial to evaluate is bounded by n degree, and we define $L = \log n$ (i.e., the total number of FFT levels). Then, the forward FFT operation requires a total of L FFT levels, where each l-th level requires the evaluation of 2^{L-l} polynomials at 2^l distinct X values. Therefore, the total number of FFT computations for the forward FFT is: $\log(n) \cdot (2^{L-l} \cdot 2^l) = 2^L \log n = n \log n$. Therefore, the time complexity of the forward FFT is $O(n \log n)$.

Using the FFT technique, we reduce the number of x points to evaluate to half as the level decreases (while the number of polynomials to evaluate doubles), and their growth and reduction cancel each other, resulting in $O(n)$ for each level. Since there are $\log n$ such levels, the total time complexity is $O(n \log n)$. The core enabler of this optimization is the special property of the x evaluation coordinates: its power (i.e., ω^i) is cyclic. To enforce this cyclic property, FFT requires the evaluation points of x to be the odd-powered primitive $2n$-th roots of unity.

A-16.3 Point-wise Multiplication

Once we have applied the forward FFT operation (**??**) to polynomial $A(X)$ and $B(X)$ as $\vec{y}^{\langle a \rangle}$ and $\vec{y}^{\langle b \rangle}$, computing the point-value representation of $C(X) = A(X) \cdot B(X)$ can be done in $O(\log n)$ using the Hadamard product $\vec{y}^{\langle c \rangle} = \vec{y}^{\langle a \rangle} \odot \vec{y}^{\langle b \rangle}$ (as explained in **??**).

A-16.4 Inverse FFT (or NTT)

Once we have computed:

$C(X) : ((x_0, y_0^{\langle c \rangle}), (x_1, y_1^{\langle c \rangle}), \cdots (x_{n-1}, y_{n-1}^{\langle c \rangle}))$, where $y_i^{\langle c \rangle} = y_i^{\langle a \rangle} \cdot y_i^{\langle b \rangle}$

, our final step is to convert $y_i^{\langle c \rangle}$ back to $\vec{c}$, the polynomial coefficients of $C(X)$. We call this reversing operation the inverse FFT.

Given an $(n-1)$-degree polynomial $A(X) = \sum_{i=0}^{n-1} a_i X^i$, the forward FFT process is computationally equivalent to evaluating the polynomial at n distinct n-th roots of unity as follows:

$y_i^{\langle a \rangle} = A(\omega^i) = \sum_{j=0}^{n-1} a_j \cdot (\omega^i)^j = \sum_{j=0}^{n-1} a_j \cdot \omega^{ij}$

The above evaluation is equivalent to computing the following matrix-vector multiplication:

$$y_i^{\langle a \rangle} = W \cdot \vec{a}, \quad \text{where } W = \begin{bmatrix} (\omega^0)^0 & (\omega^0)^1 & \cdots & (\omega^0)^{n-1} \\ (\omega^1)^0 & (\omega^1)^1 & \cdots & (\omega^1)^{n-1} \\ \vdots & \vdots & \ddots & \vdots \\ (\omega^{n-1})^0 & (\omega^{n-1})^1 & \cdots & (\omega^{n-1})^{n-1} \end{bmatrix}, \quad \vec{a} = (a_0, a_1, \cdots, a_{n-1})$$

We denote each element of W as: $(W)_{i,j} = \omega^{ij}$. The inverse FFT is a process of reversing the above computation. For this inversion, our goal is to find an inverse matrix W^{-1} such that $W^{-1} \cdot y_i^{\langle a \rangle} = W^{-1} \cdot (W \cdot \vec{a}) = (W^{-1} \cdot W) \cdot \vec{a} = I_n \cdot \vec{a} = \vec{a}$. As a solution, we propose the inverse matrix W^{-1} as follows: $(W^{-1})_{j,k} = n^{-1} \cdot \omega^{-jk}$. Now, we will show why $W^{-1} \cdot W = I_n$.

Each element of $W^{-1} \cdot W$ is computed as:
$(W^{-1} \cdot W)_{j,i} = \sum_{k=0}^{n-1} (n^{-1} \cdot \omega^{-jk} \cdot \omega^{ki}) = n^{-1} \cdot \sum_{k=0}^{n-1} \omega^{-jk} \cdot \omega^{ki} = n^{-1} \cdot \sum_{k=0}^{n-1} \omega^{(i-j)\cdot k}$
In order for $W^{-1} \cdot W$ to be I_n, the following should hold:

$$(W^{-1} \cdot W)_{j,i} = \begin{cases} 1 \text{ if } j = i \\ 0 \text{ if } j \neq i \end{cases}$$

If $j = i$, the above condition holds, because $(W^{-1} \cdot W)_{j,i} = n^{-1} \cdot \sum_{k=0}^{n-1} \omega^{(0)\cdot k} = n^{-1} \cdot \sum_{k=0}^{n-1} 1 = n^{-1} \cdot n = 1$.

In the case of $j \neq i$, we will leverage the Geometric Sum formula $\sum_{i=0}^{n-1} x^i = \frac{x^n - 1}{x - 1}$ (the proof is provided below):

⟨Theorem ??⟩ Geometric Sum Formula

Let the geometric sum $S_n = 1 + x + x^2 + \cdots + x^{n-1}$
Then, $x \cdot S_n = x + x^2 + x^3 + \cdots + x^n$
$x \cdot S_n - S_n = (x + x^2 + x^3 + \cdots + x^n) - (1 + x + x^2 + \cdots + x^{n-1}) = x^n - 1$
$S_n \cdot (x - 1) = x^n - 1$
$S_n = \frac{x^n - 1}{x - 1}$ # with the constraint that $x \neq 1$

Our goal is to compute $\sum_{k=0}^{n-1} \omega^{(i-j)\cdot k}$. Leveraging the Geometric Sum formula $\sum_{i=0}^{n-1} x^i = \frac{x^n - 1}{x - 1}$,
$\sum_{k=0}^{n-1} \omega^{(i-j)\cdot k} = \frac{(\omega^{i-j})^n - 1}{\omega^{i-j} - 1}$
$= \frac{(\omega^n)^{i-j} - 1}{\omega^{i-j} - 1}$
$= \frac{(1)^{i-j} - 1}{\omega^{i-j} - 1}$ # since the order of ω is n
Here, the denominator can't be 0, since $i \neq j$ and $-n < i - j < n$ (as $0 \leq i < n$ and $0 \leq j < n$)
$= 0$

Thus, $(W^{-1} \cdot W)_{j,i}$ is 1 if $j = i$, and 0 if $j \neq i$. Therefore, the inverse FFT can be computed as follows:

$\vec{c} = W^{-1} \cdot y_i^{\langle c \rangle}$, where $c_i = n^{-1} \cdot \sum_{j=0}^{n-1} y_i^{\langle c \rangle} \cdot \omega^{-ij}$

The above inverse FFT computation is equivalent to evaluating the polynomial $\hat{C}(X) = \sum_{i=0}^{n-1} y_i^{\langle c \rangle} \cdot X^i$ at $X = \{\omega^0, \omega^{-1}, \omega^{-2}, \cdots, \omega^{-(n-1)}\}$, which is equivalent to evaluating $\hat{C}(X)$ at $X = \{\omega^0, \omega^{n-1}, \omega^{n-2}, \cdots, \omega^1\}$. For this evaluation, we can use the same optimization technique explained in the forward FFT process (**??**) whose time complexity is $O(n \log n)$.

Part II
Post-quantum Cryptography

This chapter explains lattice-based cryptographic schemes: LWE cryptosystem, RLWE cryptosystem, GLWE cryptosystem, GLev cryptosystem, and GGSW cryptosystem. These are the essential building blocks for FHE schemes.

Required Background

- ??: ??
- ??: ??
- ??: ??

B-1 Lattice-based Cryptography

- Reference: Lattice-Based Cryptography and the Learning with Errors Problem [**?**]

Lattice-based cryptography is often considered as post-quantum cryptography, resistant against quantum computer attacks. This section describes the mathematical hard problem that is the basis of the lattice-based cryptosystems we will explore: LWE (Learning with Error) cryptosystem, RLWE (Ring Learning with Error) cryptosystem, GLWE (General Learning with Error) cryptosystem, GLev cryptosystem, and GGSW cryptosystem.

B-1.1 Overview

Suppose we have a single unknown k-dimensional vector $\vec{s}$ as a secret key, many publicly known k-dimensional vectors $\vec{a}^{\langle i \rangle}$.

And suppose we have a large set of the following dot products $\vec{s} \cdot \vec{a}^{\langle i \rangle}$:

$$\vec{s} \cdot \vec{a}^{\langle 0 \rangle} = s_0 \cdot a_0^{\langle 0 \rangle} + s_1 \cdot a_1^{\langle 0 \rangle} + \cdots + s_{k-1} \cdot a_{k-1}^{\langle 0 \rangle} = b^{\langle 0 \rangle}$$
$$\vec{s} \cdot \vec{a}^{\langle 1 \rangle} = s_0 \cdot a_0^{\langle 1 \rangle} + s_1 \cdot a_1^{\langle 1 \rangle} + \cdots + s_{k-1} \cdot a_{k-1}^{\langle 1 \rangle} = b^{\langle 1 \rangle}$$
$$\vec{s} \cdot \vec{a}^{\langle 2 \rangle} = s_0 \cdot a_0^{\langle 2 \rangle} + s_1 \cdot a_1^{\langle 2 \rangle} + \cdots + s_{k-1} \cdot a_{k-1}^{\langle 2 \rangle} = b^{\langle 2 \rangle}$$
$$\vdots$$

Suppose that all $(\vec{a}^{\langle i \rangle}, b^{\langle i \rangle})$ tuples are publicly known. An attacker only needs k such tuples to derive the secret vector $\vec{s}$. Specifically, as there are k unknown variables (i.e., $s_0, s_1, \cdots, s_{k-1}$), the attacker can solve for those k variables with k equations by using linear algebra.

However, suppose that in each equation above, we randomly add an unknown small noise $e^{\langle i \rangle}$ (i.e., error) as follows:

$$\vec{s} \cdot \vec{a}^{\langle 0 \rangle} = s_0 \cdot a_0^{\langle 0 \rangle} + s_1 \cdot a_1^{\langle 1 \rangle} + \cdots + s_{k-1} \cdot a_{k-1}^{\langle 0 \rangle} + e^{\langle 0 \rangle} \approx b^{\langle 0 \rangle}$$
$$\vec{s} \cdot \vec{a}^{\langle 1 \rangle} = s_0 \cdot a_0^{\langle 1 \rangle} + s_1 \cdot a_1^{\langle 1 \rangle} + \cdots + s_{k-1} \cdot a_{k-1}^{\langle 1 \rangle} + e^{\langle 1 \rangle} \approx b^{\langle 1 \rangle}$$
$$\vec{s} \cdot \vec{a}^{\langle 2 \rangle} = s_0 \cdot a_0^{\langle 2 \rangle} + s_1 \cdot a_1^{\langle 2 \rangle} + \cdots + s_{k-1} \cdot a_{k-1}^{\langle 2 \rangle} + e^{\langle 2 \rangle} \approx b^{\langle 2 \rangle}$$
$$\vdots$$

Then, even if the attacker has a sufficient number of $(\vec{a}^{\langle i \rangle}, b^{\langle i \rangle})$ tuples, it is not feasible to derive $s_0, s_1, \cdots, s_{k-1}$, because even a small amount of noise added to each equation prevents the linear-algebra-based direct derivation of the unknown variables. For each of the above equations, the attacker has to consider as many possibilities as there are possible values of $e^{\langle i \rangle}$. For example, if there are r possible values for each noise $e^{\langle i \rangle}$, the attacker's brute-force search space for applying linear algebra to those n equations is: $\overbrace{r \times r \times r \times \cdots \times r}^{\text{n times}} = r^n$. Thus, the number of noisy equations grows, and the aggregate possibilities of $e^{\langle i \rangle}$s grow exponentially, which means that the attacker's cost of attack grows exponentially.

Based on this intuition, the mathematical hard problem that constitutes lattice-based cryptography is as follows:

⟨Summary ??⟩ The LWE (Learning with Errors) and RLWE Problems

LWE Problem

Suppose we have a plaintext number m to encrypt. The encryption formula is: $b = \vec{s} \cdot \vec{a} + \Delta \cdot m + e$. In this formula, e is a small noise, and Δ is a scaling factor to bit-wise separate the plaintext m from the noise e with enough gap when they are added up (this is needed for successful decryption, which will be explained later).

For each encryption, a random k-dimensional vector $\vec{a} \in \mathbb{Z}_q^k$ and a small noise value $e \in \mathbb{Z}_q$ are newly sampled from $\{0, 1, \cdots, q-1\}$, where q is the ciphertext domain size. On the other hand, the k-dimensional secret vector $\vec{s}$ is the same for all encryptions. Suppose we have an enough number of ciphertext tuples:

$(\vec{a}^{\langle 1 \rangle}, b^{\langle 1 \rangle})$, where $b^{\langle 1 \rangle} = \vec{s} \cdot \vec{a}^{\langle 1 \rangle} + \Delta m^{\langle 1 \rangle} + e^{\langle 1 \rangle}$

$(\vec{a}^{\langle 2 \rangle}, b^{\langle 2 \rangle})$, where $b^{\langle 2 \rangle} = \vec{s} \cdot \vec{a}^{\langle 2 \rangle} + \Delta m^{\langle 2 \rangle} + e^{\langle 2 \rangle}$

$(\vec{a}^{\langle 3 \rangle}, b^{\langle 3 \rangle})$, where $b^{\langle 3 \rangle} = \vec{s} \cdot \vec{a}^{\langle 3 \rangle} + \Delta m^{\langle 3 \rangle} + e^{\langle 3 \rangle}$

$\vdots$

Suppose that the attacker has a sufficiently large number of $(\vec{a}^{\langle i \rangle}, b^{\langle i \rangle})$ tuples. Given this setup, the following hard problems constitute the defense mechanism of the LWE (Learning with Errors) cryptosystem:

- **Search-Hard Problem:** There is no efficient algorithm for the attacker to find out the secret key vector $\vec{s}$.
- **Decision-Hard Problem:** We create a black box system which can be configured to one of the following two modes: (1) all $b^{\langle i \rangle}$ values are purely randomly generated; (2) all $b^{\langle i \rangle}$ values are computed as the results of $\vec{s} \cdot \vec{a}^{\langle i \rangle} + e^{\langle i \rangle}$ based on the randomly picked known public keys $\vec{a}^{\langle i \rangle}$, randomly picked unknown noises $e^{\langle i \rangle}$, and a constant unknown secret vector $\vec{s}$. Given an enough number of $(\vec{a}^{\langle i \rangle}, b^{\langle i \rangle})$ tuples generated by this black box system, the attacker has no efficient algorithm to determine which mode this black box system is configured to.

These two problems are interchangeable.

RLWE Problem

In the case of the RLWE (Ring Learning with Errors) problem, the only difference is that $\vec{a}$, b, $\vec{s}$, m, and e are replaced by polynomials $(n-1)$-degree polynomials A, B, S, M, and E in $\mathbb{Z}_q[X]/(x^n + 1)$, and its search-hard problem is finding the unknown n coefficients of the secret polynomial S.

B-1.2 LWE Cryptosystem

The LWE cryptosystem uses the following encryption formula: $b = \vec{s} \cdot \vec{a} + \Delta \cdot m + e$ (where $\vec{s}$ is a secret key, $\vec{a}$ is a public key randomly picked per encryption, m is a plaintext, e is small noise randomly picked per encryption from a normal distribution, and b is a ciphertext). Δ is a scaling factor of the plaintext M (shifting m by $\log_2 \Delta$ bits to the left). Before encrypting the plaintext, we left-shift the plaintext several bits (i.e., $\log_2 \Delta$ bits) to secure sufficient space to store the error in the lower bits.

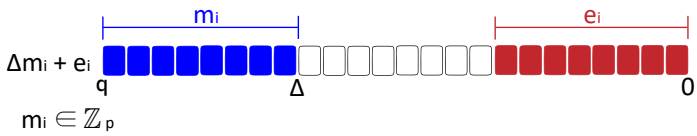


Figure 8: An illustration of LWE's plaintext scaling and adding a noise: $\Delta \cdot m + e \in \mathbb{Z}_q$

?? visually illustrates the term $\Delta \cdot m + e$, where the plaintext m left-shifted by $\log_2\Delta$ bits and noised by the noise e. The actual encryption and decryption formulas are as follows:

⟨Summary ??⟩ Lattice-based LWE Cryptosystem

- **Encryption:** $b^{\langle i \rangle} = \vec{s} \cdot \vec{a}^{\langle i \rangle} + \Delta \cdot m^{\langle i \rangle} + e^{\langle i \rangle}$, where $b^{\langle i \rangle}$ and $\vec{a}^{\langle i \rangle}$ are publicly known also to the attacker, while $\vec{s}, m^{\langle i \rangle}, e^{\langle i \rangle}$ are unknown (only known by the secret key owner).
- **Decryption:** $\frac{\lceil b^{\langle i \rangle} - \vec{s} \cdot \vec{a}^{\langle i \rangle} \rfloor_\Delta}{\Delta} = \frac{\lceil \Delta m^{\langle i \rangle} + e^{\langle i \rangle} \rfloor_\Delta}{\Delta} = m^{\langle i \rangle} \left(\text{provided } e^{\langle i \rangle} < \frac{\Delta}{2} \right)$

$\lfloor \rceil_\Delta$ means rounding the number to the nearest multiple of Δ. For example, $\lfloor 16 \rceil_{10} = 20$, which is rounding 16 to the nearest multiple of 10. As another example, $\lfloor 17 \rceil_8 = 16$, which rounds 17 to the nearest multiple of 8 (note that 17 is closer to 16 than to 24; thus, it is rounded to 16).

Correctness: In the decryption scheme, computing $b^{\langle i \rangle} - \vec{s} \cdot \vec{a}^{\langle i \rangle}$ gives $\Delta \cdot m^{\langle i \rangle} + e^{\langle i \rangle}$, which is **??**. Then, $\lceil \Delta \cdot m^{\langle i \rangle} + e^{\langle i \rangle} \rfloor_\Delta$ (i.e., rounding the value to the nearest multiple of Δ) gives $\Delta \cdot m^{\langle i \rangle}$, provided the added noise $e^{\langle i \rangle} < \frac{\Delta}{2}$. That is, if the noise is less than $\frac{\Delta}{2}$, it will disappear during the rounding. Finally, right-shifting $\Delta \cdot m^{\langle i \rangle}$ by $\log_2\Delta$ bits gives $m^{\langle i \rangle}$. To summarize, if we ensure $e^{\langle i \rangle} < \frac{\Delta}{2}$ (which is why the noise $e^{\langle i \rangle}$ should be smaller than this threshold), then we can eliminate $e^{\langle i \rangle}$ during the decryption's rounding process and retrieve the original $\Delta \cdot m^{\langle i \rangle}$. The reason we scaled $m^{\langle i \rangle}$ by Δ is to: (i) create space for storing $e^{\langle i \rangle}$ in the lower bits during encryption such that the noise bits do not interfere with the plaintext bits (to avoid corrupting the plaintext bits); and (ii) blow away the noise $e^{\langle i \rangle}$ stored in the lower bits during decryption without corrupting the plaintext $m^{\langle i \rangle}$.

Security: Given that an attacker has a large list of $(\vec{a}^{\langle i \rangle}, b^{\langle i \rangle})$ (i.e., many ciphertexts), it is almost impossible for them to derive $\vec{s}$, due to the random noise $e^{\langle i \rangle}$ added in each encryption (which is a search-hard problem described in **??**). This is because even small added unknown noises $e^{\langle i \rangle}$ greatly change the mathematical solution for $\vec{s}$ that satisfies all the $b^{\langle i \rangle} = \vec{s} \cdot \vec{a}^{\langle i \rangle} + \Delta \cdot m^{\langle i \rangle} + e^{\langle i \rangle}$ equations.

Even in the case that the attacker has a large list of $(\vec{a}^{(j)}, b^{(j)})$ generated for the same ciphertext $m^{\langle i \rangle}$ (where each ciphertext used different $\vec{a}^{(j)}$ and $e^{(j)}$ to encrypt the same $m^{\langle i \rangle}$), he still cannot derive $m^{\langle i \rangle}$, because a randomly picked different noise $e^{(j)}$ is used for every $(\vec{a}^{(j)}, b^{(j)})$ and is accumulated over ciphertexts, which exponentially complicates the difficulty of the linear algebra involved in solving $\vec{s}$. Also, in the actual cryptosystem (**??**), the public key $\vec{a}^{\langle i \rangle}$ and the secret key $\vec{s}$ are not a single number but a long vector comprising many random numbers. Thus, adding $A^{\langle i \rangle} \cdot \vec{s}$ to $\Delta m^{\langle i \rangle} + e^{\langle i \rangle}$ increases the entropy of randomness against the attack.

To summarize, lattice-based cryptography hides plaintext by adding the encryption component $\vec{s} \cdot \vec{a}$ to it, along with a small random noise e. During decryption, the secret key owner re-creates this encryption component $\vec{a} \cdot \vec{s}$ by using her $\vec{s}$ and removes it. She then removes the noise e using

the rounding technique and finally right-shifts the remaining Δm by $\log_2\Delta$ bits to get m.

B-1.3 RLWE Cryptosystem

In the RLWE cryptosystem, the formula in ⟨Summary **??**⟩ is the same, but $\vec{s}, \vec{a}^{\langle i\rangle}, b^{\langle i\rangle}, m^{\langle i\rangle}, e^{\langle i\rangle}$ are replaced by polynomials $S, A^{\langle i\rangle}, B^{\langle i\rangle}, M^{\langle i\rangle}, E^{\langle i\rangle}$ as follows:

⟨Summary ??⟩ Lattice-based RLWE Cryptosystem

- **Encryption:** $B^{\langle i\rangle} = S \cdot A^{\langle i\rangle} + \Delta \cdot M^{\langle i\rangle} + E^{\langle i\rangle}$, where $B^{\langle i\rangle}$ and $A^{\langle i\rangle}$ are publicly known also to the attacker, while $S, m^{\langle i\rangle}, E$ are unknown (only known by the secret key owner).
- **Decryption:** $\dfrac{\lceil B^{\langle i\rangle} - S \cdot A^{\langle i\rangle} \rfloor_\Delta}{\Delta} = \dfrac{\lceil \Delta M^{\langle i\rangle} + E^{\langle i\rangle} \rfloor_\Delta}{\Delta} = M^{\langle i\rangle} \left(\text{provided } E^{\langle i\rangle} < \dfrac{\Delta}{2} \right)$

  $\lceil \rfloor_\Delta$ is equivalent to rounding each term's coefficient in the polynomial.

B-2 LWE Cryptosystem

- Reference: TFHE Deep Dive: Part I - Ciphertext types [**?**]

B-2.1 Setup

Let $[0, t-1)$ be the plaintext range, and $[0, q-1)$ the ciphertext range, where $t < q$ (t is much smaller than q). Randomly pick a vector $\vec{s}$ of length k comprising k binary numbers as a secret key (denoted as $S \xleftarrow{\$} \mathbb{Z}_2^k$). Let $\Delta = \frac{q}{t}$, the scaling factor of plaintext.

B-2.2 Encryption

1. Suppose we have a plaintext $m \in \mathbb{Z}_t$ to encrypt.
2. Randomly pick a vector $\vec{a} \in \mathbb{Z}_q^k$ (of length k) as a one-time public key (denoted as $\vec{a} \xleftarrow{\$} \mathbb{Z}_q^k$).
3. Randomly pick a small one-time noise $e \in \mathbb{Z}_q$ sampled from the Gaussian distribution χ_σ (denoted as $e \xleftarrow{\chi_\sigma} \mathbb{Z}_q$).
4. Scale m by Δ, which is to compute $\Delta \cdot m$. This converts $m \in \mathbb{Z}_t$ into $\Delta \cdot m \in \mathbb{Z}_q$.
5. Compute $b = \vec{a} \cdot \vec{s} + \Delta \cdot m + e \in \mathbb{Z}_q$.

The LWE encryption formula is summarized as follows:

⟨Summary ??⟩ LWE Encryption

Initial Setup: $\Delta = \frac{q}{t}$, $\vec{s} \xleftarrow{\$} \mathbb{Z}_2^k$, where t divides q

Encryption Input: $m \in \mathbb{Z}_t$, $\vec{a} \xleftarrow{\$} \mathbb{Z}_q^k$, $e \xleftarrow{\chi_\sigma} \mathbb{Z}_q$

1. Scale up $m \longrightarrow \Delta \cdot m \ \in \mathbb{Z}_q$
2. Compute $b = \vec{a} \cdot \vec{s} + \Delta m + e \pmod q$
3. $\mathsf{LWE}_{\vec{s},\sigma}(\Delta m) = (\vec{a}, b) \ \in \mathbb{Z}_q^{k+1}$

B-2.3 Decryption

1. Given the ciphertext $(\vec{a}, b)$ where $b = \vec{a} \cdot \vec{s} + \Delta \cdot m + e \in \mathbb{Z}_q$, compute $b - \vec{a} \cdot \vec{s}$, which gives the same value as $\Delta \cdot m + e \in \mathbb{Z}_q$.
2. Round $\Delta \cdot m + e \in \mathbb{Z}_q$ to the nearest multiple of Δ (i.e., round it as a base Δ number), which is denoted as $\lceil \Delta \cdot m + e \rfloor_\Delta$. This rounding operation successfully eliminates e and gives Δm, provided e is small enough to be $e < \frac{\Delta}{2}$. If $e \geq \frac{\Delta}{2}$, then some of the higher bits of the noise e will overlap with the plaintext m, won't be blown away, and will corrupt some lower bits of the plaintext m.

3. Compute $\frac{\Delta m}{\Delta}$, which is equivalent to right-shifting $\lceil \Delta \cdot m + e \rfloor_\Delta$ by $\log_2 \Delta$ bits.

The LWE decryption formula is summarized as follows:

⟨Summary ??⟩ LWE Decryption

Decryption Input: $\mathsf{ct} = (\vec{a}, b) \in \mathbb{Z}_q^{k+1}$

1. $\mathsf{LWE}_{S,\sigma}^{-1}(\mathsf{ct}) = b - \vec{a} \cdot \vec{s} = \Delta m + e \pmod q$
2. Scale down $\left\lceil \frac{\Delta m + e}{\Delta} \right\rfloor \bmod t = m \in \mathbb{Z}_t$

For correct decryption, the noise e should be $e < \frac{\Delta}{2}$.

During decryption, the secret key owner can subtract $\vec{a} \cdot \vec{s}$ from b because he can directly compute $\vec{a} \cdot \vec{s}$ by using his secret key $\vec{s}$.

The reason we scaled the plaintext m by Δ is: (i) to left-shift m by $\log_2 \Delta$ bits and separate it from the noise e in the lower bits during encryption, whereas e is essential to make it hard for the attacker to guess m or $\vec{s}$; and (ii) to eliminate e in the lower bits by right-shifting it by Δ bits without compromising m in the higher bits during decryption. The process of right-shifting (i.e., scaling) the plaintext m by $\log_2 \Delta$ bits, followed by adding the noise e, is illustrated in **??**.

B-2.3.1 In the Case of t not Dividing q

In Summary **??** (**??**), we assumed that t divides q. In this case, there is no upper or lower limit on the size of plaintext m: its value is allowed to wrap around modulo t indefinitely, yet the decryption works correctly. This is because any m value greater than t will be correctly modulo-reduced by t when we do modulo reduction by q during decryption.

On the other hand, suppose that t does not divide q. In such a case, we set the scaling factor as $\Delta = \left\lfloor \frac{q}{t} \right\rfloor$. Then, provided $q \gg t$, the decryption works correctly even if m is a large value that wraps around t. We will show why this is so.

We can denote plaintext $m \bmod t$ as $m = m' + kt$, where $m' \in \mathbb{Z}_t$ and k are integers used to represent the modulo t wrap-around value portion of m. And we set the plaintext scaling factor as $\Delta = \left\lfloor \frac{q}{t} \right\rfloor$. Then, the noise-added scaled plaintext value is as follows:

$$\left\lfloor \frac{q}{t} \right\rfloor \cdot m + e = \left\lfloor \frac{q}{t} \right\rfloor \cdot m' + \left\lfloor \frac{q}{t} \right\rfloor \cdot kt + e \ \# \text{ by applying } m = m' + kt$$
$$= \left\lfloor \frac{q}{t} \right\rfloor \cdot m' + \frac{q}{t} \cdot kt - (\frac{q}{t} - \left\lfloor \frac{q}{t} \right\rfloor) \cdot kt + e \ \# \text{ where } 0 \leq \frac{q}{t} - \left\lfloor \frac{q}{t} \right\rfloor < 1$$
$$= \left\lfloor \frac{q}{t} \right\rfloor \cdot m' + qk - (\frac{q}{p} - \left\lfloor \frac{q}{t} \right\rfloor) \cdot kt + e$$

We treat the above noisy scaled ciphertext as $= \left\lfloor \frac{q}{t} \right\rfloor \cdot m' + qk - e' + e$, where $e' = kt$ is the maximum possible value of $\left\lfloor \frac{q}{t} \right\rfloor) \cdot kt$. In other words, we overestimate the noise caused by $(\frac{q}{p} - \left\lfloor \frac{q}{t} \right\rfloor) \cdot kt$ to

kt, because the maximum value this term can attain is less than kt.

Given the LWE decryption relation $b - \vec{a} \cdot \vec{s} \bmod q = \Delta m + e$, we can decrypt the above message by performing:

$$\left\lceil \frac{1}{\lfloor \frac{q}{t} \rfloor} \cdot \left(\left\lfloor \frac{q}{t} \right\rfloor \cdot m' + qk - e' + e \bmod q \right) \right\rfloor \bmod t$$

$$= \left\lceil \frac{1}{\lfloor \frac{q}{t} \rfloor} \cdot \left(\left\lfloor \frac{q}{t} \right\rfloor \cdot m' + qk - kt + e \bmod q \right) \right\rfloor \bmod t$$

$$= \left\lceil \frac{1}{\lfloor \frac{q}{t} \rfloor} \cdot \left(\left\lfloor \frac{q}{t} \right\rfloor \cdot m' - kt + e \right) \right\rfloor \bmod t$$

$$= \left\lceil m' - \frac{kt + e}{\lfloor \frac{q}{t} \rfloor} \right\rfloor \bmod t$$

$$= m' \ \# \text{ provided } \frac{kt + e}{\lfloor \frac{q}{t} \rfloor} < \frac{1}{2}$$

To summarize, if we set the plaintext's scaling factor as $\Delta = \left\lfloor \frac{q}{t} \right\rfloor$, where t does not divide q, the decryption works correctly as long as the error bound $\frac{kt + e}{\lfloor \frac{q}{t} \rfloor} < \frac{1}{2}$ holds. This error bound can break if: (1) the noise e is too large; (2) the plaintext modulus t is too large; and (3) the plaintext value wraps around t too many times (i.e., k is too large). A solution to ensure $\frac{kt + e}{\lfloor \frac{q}{t} \rfloor} < \frac{1}{2}$ is that the ciphertext modulus q is sufficiently large. To put it differently, if $q \gg t$, then the error bound will hold.

Therefore, we can generalize the formula for the plaintext's scaling factor in Summary **??** (in **??**) as $\left\lfloor \frac{q}{t} \right\rfloor$ where t does not necessarily divide q.

B-3 RLWE Cryptosystem

The RLWE cryptosystem's ciphertext is a tuple (A, B), where $B = S \cdot A + \Delta \cdot M + E$. The public key A and the secret key S are $(n-1)$-degree polynomials. The message M and the noise E are an $(n-1)$-degree polynomial, each. Like in LWE, a new public key A is created for each ciphertext, whereas the same secret key S is used for all ciphertexts. In this section, we denote each ciphertext instance as (A, b) instead of $(A^{\langle i \rangle}, b^{\langle i \rangle})$ for simplicity.

In RLWE, all polynomials are computed in the polynomial ring $\mathbb{Z}_q[x]/x^n + 1$, where $x^n + 1$ is a cyclotomic polynomial with $n = 2^f$ for some integer f and the polynomial coefficients are in $\mathbb{Z}_q$. Thus, all polynomials in RLWE have the coefficient range $\mathbb{Z}_q$ and the maximum polynomial degree of $n - 1$. For simplicity, we denote $\mathcal{R}_{\langle n,q \rangle} = \mathbb{Z}_q[x]/(x^n + 1)$.

B-3.1 Setup

Let t the size of plaintext, and q the size of ciphertext, where $t < q$ (t is much smaller than q) and $t|q$ (i.e., t divides q). Randomly pick a $(n-1)$-degree polynomial $S \in \mathcal{R}_{\langle n,2 \rangle}$ whose coefficients are binary numbers, 1 or -1, as a secret key. Let $\Delta = \left\lfloor \frac{q}{t} \right\rfloor$ the scaling factor of plaintext.

Notice that RLWE's setup parameters are similar to that of LWE. One difference is that S is not a vector of length k, but an $(n-1)$-degree polynomial encoding n secret coefficients, where each coefficient is a randomly picked binary number in $\{0, 1\}$ (denoted as $S \xleftarrow{\$} \mathcal{R}_{\langle n,2 \rangle}$).

B-3.2 Encryption

1. Suppose we have an $(n-1)$-degree polynomial $M \in \mathcal{R}_{\langle n,t \rangle}$ whose coefficients represent the plaintext numbers to encrypt.
2. Randomly pick an $(n-1)$-degree polynomial $A \in \mathcal{R}_{\langle n,q \rangle}$ as a one-time public key (denoted as $A \xleftarrow{\$} \mathcal{R}_{\langle n,q \rangle}$).
3. Randomly pick a small polynomial $E \in \mathcal{R}_{\langle n,q \rangle}$ as a one-time noise, whose n coefficients are small numbers in $\mathbb{Z}_q$ randomly sampled from the Gaussian distribution χ_σ (denoted as $E \xleftarrow{\chi_\sigma} \mathcal{R}_{\langle n,q \rangle}$).
4. Scale M by Δ, which is to compute $\Delta \cdot M$. This converts $M \in \mathcal{R}_{\langle n,t \rangle}$ into $\Delta \cdot M \in \mathcal{R}_{\langle n,q \rangle}$.
5. Compute $B = A \cdot S + \Delta \cdot M + E \bmod \mathcal{R}_{\langle n,q \rangle}$ (i.e., reduce the degree by n and the coefficient by modulo q).
6. The final ciphertext is (A, B).

The RLWE encryption formula is summarized as follows:

⟨Summary ??⟩ RLWE Encryption

Initial Setup: $\Delta = \left\lfloor \frac{q}{t} \right\rfloor$, $S \xleftarrow{\$} \mathcal{R}_{\langle n,2 \rangle}$

Encryption Input: $M \in \mathcal{R}_{\langle n,t \rangle}$, $A \xleftarrow{\$} \mathcal{R}_{\langle n,q \rangle}$, $E \xleftarrow{\chi_\sigma} \mathcal{R}_{\langle n,2 \rangle}$

1. Scale up $M \longrightarrow \Delta M \ \in \mathcal{R}_{\langle n,q \rangle}$
2. Compute $B = A \cdot S + \Delta M + E \mod \mathcal{R}_{\langle n,q \rangle}$
3. $\mathsf{RLWE}_{S,\sigma}(\Delta M) = (A, B) \ \in \mathcal{R}^2_{\langle n,q \rangle}$

B-3.3 Decryption

1. Given the ciphertext (A, B) where $B = A \cdot S + \Delta \cdot M + E \in \mathcal{R}_{\langle n,q \rangle}$, compute $B - A \cdot S = \Delta \cdot M + E$.
2. Round each coefficient of the polynomial $\Delta \cdot M + E \in \mathcal{R}_{\langle n,q \rangle}$ to the nearest multiple of Δ (i.e., round it as a base Δ number), which is denoted as $\lceil \Delta \cdot M + E \rfloor_\Delta$. This rounding operation successfully eliminates E and gives $\Delta \cdot M$. One caveat is that the noise E's each coefficient e_i should be small enough to be $e_i < \dfrac{\Delta}{2}$ in order to be eliminated during the rounding. Otherwise, some of e_i's higher bits will overlap and corrupt the plaintext m_i coefficient's lower bits and won't be blown away.
3. Compute $\dfrac{\Delta \cdot M}{\Delta}$, which is equivalent to right-shifting each polynomial coefficient in $\Delta \cdot M$ by $\log_2 \Delta$ bits.

In summary, the RLWE decryption formula is summarized as follows:

⟨Summary ??⟩ RLWE Decryption

Decryption Input: $C = (A, B) \ \in \mathbb{R}^2_{\langle n,r \rangle}$

1. $\mathsf{RLWE}^{-1}_{S,\sigma}(C) = B - A \cdot S = \Delta M + E \ (\mathrm{mod} \mathcal{R}_{\langle n,q \rangle})$
2. Scale down $\left\lceil \dfrac{\Delta M + E}{\Delta} \right\rfloor \mod t = M \ \in \mathcal{R}_{\langle n,t \rangle}$

For correct decryption, every noise coefficient e_i of polynomial E should be: $e_i < \dfrac{\Delta}{2}$. And in case t does not divide q, q should be sufficiently larger than t.

B-4 GLWE Cryptosystem

The GLWE cryptosystem is a generalized form to encompass both the LWE and RLWE cryptosystems. The GLWE cryptosystem's ciphertext is a tuple $(\{A_i\}_{i=0}^{k-1}, B)$, where $B = \sum_{i=0}^{k-1} (A_i \cdot S_i) + \Delta \cdot M + E$. The public key $\{A_i\}_{i=0}^{k-1}$ and the secret key $\{S_i\}_{i=0}^{k-1}$ are a list of k $(n-1)$-degree polynomials, each. The message M and the noise E are an $(n-1)$-degree polynomial, each. Like in LWE and GLWE, a new public key A is created for each ciphertext, whereas the same secret key S is used for all ciphertexts. In this section, we denote each ciphertext instance as $(\{A_i\}_{i=0}^{k-1}, B)$ instead of $(\{A_i\}_{i=0}^{k-1,\langle j \rangle}, B^{\langle j \rangle})$ for simplicity.

B-4.1 Setup

Let t the size of plaintext, and q the size of ciphertext, where $t < q$ (t is much smaller than q) and $t|q$ (i.e., t divides q). Randomly pick a list of k $(n-1)$-degree polynomials as a secret key, where each polynomial coefficient is a randomly picked binary number in $\{0, 1\}$ (i.e., $\{S_i\}_{i=0}^{k-1} \xleftarrow{\$} \mathcal{R}_{\langle n,2 \rangle}^k$). Let $\Delta = \left\lfloor \frac{q}{t} \right\rfloor$ the scaling factor of plaintext.

Notice that GLWE's setup parameters are similar to that of RLWE. One difference is that S is not an $(n-1)$-degree polynomial encoding n secret coefficients, but a list of k such $(n-1)$-degree polynomials encoding total $n \cdot k$ secret coefficients.

B-4.2 Encryption

Suppose we have an $(n-1)$-degree polynomial $M \in \mathcal{R}_{(n,t)}$ whose coefficients represent the plaintext numbers to encrypt.

1. Randomly pick a list of k $(n-1)$-degree polynomials $\{A_i\}_{i=0}^{k-1} \xleftarrow{\$} \mathcal{R}_{\langle n,q \rangle}^k$ as a one-time public key.
2. Randomly pick a small polynomial $E \xleftarrow{\chi_\sigma} \mathcal{R}_{\langle n,q \rangle}$ as a one-time noise, whose n coefficients are small numbers in $\mathbb{Z}_q$ randomly sampled from the Gaussian distribution χ_σ.
3. Scale M by Δ, which is to compute $\Delta \cdot M$. This converts $M \in \mathcal{R}_{\langle n,t \rangle}$ into $\Delta \cdot M \in \mathcal{R}_{\langle n,q \rangle}$.
4. Compute $B = \sum_{i=0}^{k-1} (A_i \cdot S_i) + \Delta \cdot M + E \in \mathcal{R}_{\langle n,q \rangle}$.

The GLWE encryption formula is summarized as follows:

⟨Summary ??⟩ GLWE Encryption

Initial Setup: $\Delta = \left\lfloor \frac{q}{t} \right\rfloor$, $\{S_i\}_{i=0}^{k-1} \xleftarrow{\$} \mathcal{R}_{\langle n,2 \rangle}^k$

Encryption Input: $M \in \mathcal{R}_{\langle n,t \rangle}$, $\{A_i\}_{i=0}^{k-1} \xleftarrow{\$} \mathcal{R}_{\langle n,q \rangle}^k$, $E \xleftarrow{\chi_\sigma} \mathcal{R}_{\langle n,q \rangle}$

1. Scale up $M \longrightarrow \Delta M \ \in \mathcal{R}_{\langle n,q \rangle}$

2. Compute $B = \sum_{i=0}^{k-1} (A_i \cdot S_i) + \Delta M + E \ \in \mathcal{R}_{\langle n,q \rangle}$
3. $\mathsf{GLWE}_{S,\sigma}(\Delta M) = (\{A_i\}_{i=0}^{k-1}, B) \ \in \mathcal{R}_{\langle n,q \rangle}^{k+1}$

B-4.3 Decryption

1. Given the ciphertext $(\{A_i\}_{i=0}^{k-1}, B)$ where $B = \sum_{i=0}^{k-1} (A_i \cdot S_i) + \Delta \cdot M + E \in \mathcal{R}_{\langle n,q \rangle}$, compute $B - \sum_{i=0}^{k-1} (A_i \cdot S_i) = \Delta \cdot M + E$.
2. Round each coefficient of the polynomial $\Delta \cdot M + E \in \mathcal{R}_{\langle n,q \rangle}$ to the nearest multiple of Δ (i.e., round it as a base Δ number), which is denoted as $\lceil \Delta \cdot M + E \rfloor_\Delta$. This operation successfully eliminates E and gives $\Delta \cdot M$. One caveat is that E's each coefficient e_i has to be $e_i < \frac{\Delta}{2}$ to be eliminated during the rounding. Otherwise, some of e_i's higher bits will overlap the plaintext m_i coefficient's lower bit and won't be eliminated during decryption, corrupting the plaintext m_1.
3. Compute $\frac{\Delta \cdot M}{\Delta}$, which is equivalent to right-shifting each polynomial coefficient in $\Delta \cdot M$ by $\log_2 \Delta$ bits.

In summary, the GLWE decryption formula is summarized as follows:

⟨Summary ??⟩ GLWE Decryption

Decryption Input: $C = (\{A_i\}_{i=0}^{k-1}, B) \ \in \mathcal{R}_{\langle n,q \rangle}^{k+1}$

1. $\mathsf{GLWE}_{S,\sigma}^{-1}(C) = B - \sum_{i=0}^{k-1} (A_i \cdot S_i) = \Delta M + E \ \in \mathcal{R}_{\langle n,q \rangle}$
2. Scale down $\left\lceil \frac{\Delta M + E}{\Delta} \right\rfloor \bmod t = M \ \in \mathcal{R}_{\langle n,t \rangle}$

For correct decryption, every noise coefficient e_i of polynomial E should be: $e_i < \frac{\Delta}{2}$.

B-4.3.1 Discussion

1. **LWE** is a special case of GLWE where the polynomial ring's degree $n = 1$. That is, all polynomials in $\{A_i\}_{i=0}^{k-1}, \{S_i\}_{i=0}^{k-1}, E$, and M are zero-degree polynomial constants. Instead, there are k such constants for A_i and S_i, so each of them forms a vector.
2. **RLWE** is a special case of GLWE where $k = 1$. That is, the secret key S is a single polynomial S_0, and each encryption is processed by only a single polynomial A_0 as a public key.
3. **Size of *n*:** A large polynomial degree n increases the number of the secret key's coefficient terms (i.e., $S_i = s_{i,0} + s_{i,1}X + \ \cdots \ + s_{i,n-1}X^{n-1}$), which makes it more difficult to guess the complete secret key. The same applies to the noise polynomial E and the public key polynomials A_i, thus making it harder to solve the search-hard problem (**??**). Also, higher-degree polynomials can

encode more plaintext terms in the same plaintext polynomial M, improving the throughput efficiency of processing ciphertexts.

4. **Size of k:** A large k increases the number of the secret key polynomials $(S_0, S_1, \cdots, S_k)$ and the number of the one-time public key polynomials $(A_0, A_1, \cdots, A_k)$, which makes it more difficult for the attacker to guess the complete secret keys. Meanwhile, there is only a single M and E polynomials per GLWE ciphertext, regardless of the size of k.
5. **Reducing the Ciphertext Size:** The public key $\{A_i\}_{i=0}^{k-1}$ has to be created for each ciphertext, which is a big size. To reduce this size, each ciphertext can instead include the seed d for the pseudo-random number generation hash function H. Then, the public key can be lively computed $k-1$ times upon each encryption & decryption as $\{H(s), H(H(s)), H(H(H(s)), \cdots\}$. Note that H, by nature, generates the same sequence of numbers given the same random initial seed d.

B-4.4 An Alternative Version of GLWE

The following is an alternative version of ⟨Summary **??**⟩, where the sign of each A_iS_i is flipped in the encryption and decryption formula as follows:

⟨Summary ??⟩ An Alternative GLWE Cryptosystem

Initial Setup: $\Delta = \left\lfloor \frac{q}{t} \right\rfloor$, $\{S_i\}_{i=0}^{k-1} \xleftarrow{\$} \mathcal{R}_{\langle n,2\rangle}^{k}$

Encryption Input: $M \in \mathcal{R}_{\langle n,t\rangle}$, $\{A_i\}_{i=0}^{k-1} \xleftarrow{\$} \mathcal{R}_{\langle n,q\rangle}^{k}$, $E \xleftarrow{\chi_\sigma} \mathcal{R}_{\langle n,q\rangle}$

1. Scale up $M \longrightarrow \Delta M \ \in \mathcal{R}_{\langle n,q\rangle}$
2. Compute $B = -\sum_{i=0}^{k-1} (A_i \cdot S_i) + \Delta M + E \ \in \mathcal{R}_{\langle n,q\rangle}$
3. $\mathsf{GLWE}_{S,\sigma}(\Delta M) = (\{A_i\}_{i=0}^{k-1}, B) \ \in \mathcal{R}_{\langle n,q\rangle}^{k+1}$

Decryption Input: $C = (\{A_i\}_{i=0}^{k-1}, B) \ \in \mathcal{R}_{\langle n,q\rangle}^{k+1}$

1. $\mathsf{GLWE}_{S,\sigma}^{-1}(C) = B + \sum_{i=0}^{k-1} (A_i \cdot S_i) = \Delta M + E \ \in \mathcal{R}_{\langle n,q\rangle}$
2. Scale down $\left\lceil \frac{\Delta M + E}{\Delta} \right\rfloor = M \ \in \mathcal{R}_{\langle n,t\rangle}$

For correct decryption, every noise coefficient e_i of polynomial E should be: $e_i < \frac{\Delta}{2}$.

Even if the A_iS_i terms flip their signs, the decryption stage cancels out those terms by adding their equivalent double-sign-flipped terms; thus, the same correctness of decryption is preserved as in the original version.

B-4.5 Public Key Encryption

The encryption scheme in **??** assumes that it is the secret key owner who encrypts each plaintext. In this section, we explain a public key encryption scheme in which we create a public key counterpart of the secret key Anyone who knows the public key can encrypt the plaintext in such a way that only the secret key owner can decrypt it. The high-level idea is that a portion of the components to be used in the encryption stage is pre-computed at the setup stage and published as a public key. At the actual encryption stage, the public key is multiplied by an additional randomness (U) and added to additional noise $(E_1, \vec{E}_2)$ to create unpredictable randomness for each encrypted ciphertext. The actual scheme is as follows:

⟨Summary ??⟩ GLWE Public Key Encryption

Initial Setup:

- The scaling factor $\Delta = \left\lfloor \frac{q}{t} \right\rfloor$
- The secret key $\vec{S} = \{S_i\}_{i=0}^{k-1} \xleftarrow{\$} \mathcal{R}_{\langle n,2\rangle}^k$
- The public key pair $(PK_1, \overrightarrow{PK_2}) \in \mathcal{R}_{\langle n,q\rangle}^{k+1}$ is generated as follows:
  $\vec{A} = \{A_i\}_{i=0}^{k-1} \xleftarrow{\$} \mathcal{R}_{\langle n,q\rangle}^k, \quad E \xleftarrow{\sigma} \mathcal{R}_{\langle n,q\rangle}$
  $PK_1 = \vec{A} \cdot \vec{S} + E \in \mathcal{R}_{\langle n,q\rangle}$
  $\overrightarrow{PK_2} = \vec{A} \in \mathcal{R}_{\langle n,q\rangle}^k$

Encryption Input: $M \in \mathcal{R}_{\langle n,t\rangle}, \quad U \xleftarrow{\$} \mathcal{R}_{\langle n,2\rangle}, \; E_1 \xleftarrow{\sigma} \mathcal{R}_{\langle n,q\rangle}, \; \vec{E}_2 \xleftarrow{\sigma} \mathcal{R}_{\langle n,q\rangle}^k$

1. Scale up $M \longrightarrow \Delta M \; \in \mathcal{R}_{\langle n,q\rangle}$
2. Compute the following:
   $B = PK_1 \cdot U + \Delta M + E_1 \; \in \mathcal{R}_{\langle n,q\rangle}$
   $\vec{D} = \overrightarrow{PK_2} \cdot U + \vec{E}_2 \in \mathcal{R}_{\langle n,q\rangle}^k$ # $\overrightarrow{PK_2} \cdot U$ multiplies U to each element of $\overrightarrow{PK_2}$
3. $\mathsf{GLWE}_{S,\sigma}(\Delta M) = (\vec{D}, B) \; \in \mathcal{R}_{\langle n,q\rangle}^{k+1}$

Decryption Input: $C = (\vec{D}, B) \; \in \mathcal{R}_{\langle n,q\rangle}^{k+1}$

1. $\mathsf{GLWE}_{S,\sigma}^{-1}(C) = B - \vec{D} \cdot \vec{S} = \Delta M + E_{all} \; \in \mathcal{R}_{\langle n,q\rangle}$
2. Scale down $\left\lceil \frac{\Delta M + E_{all}}{\Delta} \right\rfloor = M \; \in \mathcal{R}_{\langle n,t\rangle}$

For correct decryption, every noise coefficient e_i of polynomial E_{all} should be: $e_i < \frac{\Delta}{2}$.

The equation in the 1st step of the decryption process is derived as follows:

$\mathsf{GLWE}_{S,\sigma}^{-1}(\; C = (B, \vec{D}) \;) = B - \vec{D} \cdot \vec{S}$
$= (PK_1 \cdot U + \Delta M + E_1) - (\overrightarrow{PK_2} \cdot U + \vec{E}_2) \cdot \vec{S}$
$= (\vec{A} \cdot \vec{S} + E) \cdot U + \Delta M + E_1 - (\vec{A} \cdot U) \cdot \vec{S} - \vec{E}_2 \cdot \vec{S}$
$= (U \cdot \vec{A}) \cdot \vec{S} + E \cdot U + \Delta M + E_1 - (U \cdot \vec{A}) \cdot \vec{S} - \vec{E}_2 \cdot \vec{S}$
$= \Delta M + E \cdot U + E_1 - \vec{E}_2 \cdot \vec{S}$
$= \Delta M + E_{all}$ # where $E_{all} = E \cdot U + E_1 - \vec{E}_2 \cdot \vec{S}$

Security: The GLWE encryption scheme's encryption formula (Summary **??** in **??**) is as follows:
$\mathsf{GLWE}_{S,\sigma}(\Delta M) = (\ \vec{A},\ B = \vec{A} \cdot \vec{S} + \Delta M + E\ )$

, where the hardness of the LWE and RLWE problems guarantees that guessing $\vec{S}$ is difficult given $\vec{A}$ and E are randomly picked at each encryption. On the other hand, the public key encryption scheme is as follows:
$\mathsf{GLWE}_{S,\sigma}(\Delta M) = (\ \vec{D} = \overrightarrow{PK}_2 \cdot U + \vec{E}_2,\ B = PK_1 \cdot U + \Delta M + E_1\ )$

, where PK_1, $\overrightarrow{PK}_2$ are fixed and U, E_1, $\vec{E}_2$ are randomly picked at each encryption. Given the polynomial degree n is large, both schemes provide the equivalent level of hardness to solve the problem.

B-5 GLev

A GLev ciphertext is a list of GLWE ciphertexts that encrypts the list of plaintexts $\frac{q}{\beta^1}M, \frac{q}{\beta^2}M, \cdots, \frac{q}{\beta^l}M$, where M is a plaintext encoded in a polynomial. Note that each i-th GLWE ciphertext of a GLev ciphertext uses a different plaintext scaling factor, which is: $\Delta_i = \frac{q}{\beta^i}$. The structure of GLev ciphertext is visually depicted in **??**.

Note that β should be some value between t and q. Especially, t should be smaller than β because if t is greater than β, then the higher bits of M will overflow beyond q when computing $\frac{q}{\beta^1}M$.

B-5.1 Encryption

⟨Summary ??⟩ GLev Encryption

$$\mathsf{GLev}^{\beta,l}_{S,\sigma}(M) = \left\{\mathsf{GLWE}_{S,\sigma}\left(\frac{q}{\beta^i}M\right)\right\}_{i=1}^{l} \in \mathcal{R}^{(k+1)\cdot l}_{\langle n,q\rangle}$$

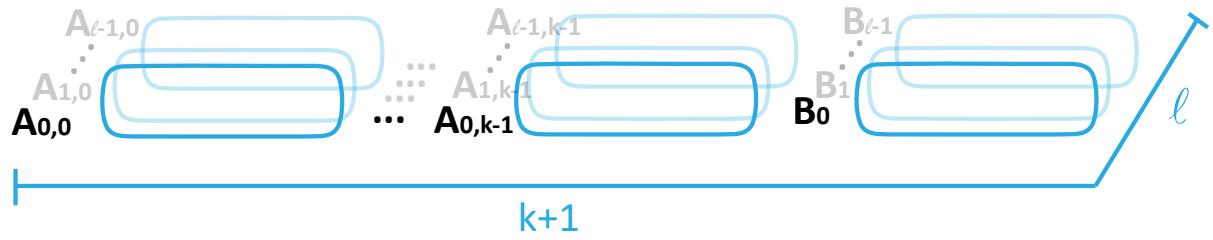


Figure 9: An illustration of a GLev ciphertext

B-5.2 Decryption

It is sufficient to decrypt any i-th GLWE ciphertext of the GLev ciphertext by using the secret S, with the scaling factor $\Delta_i = \frac{q}{\beta^i}$.

B-5.3 Lev and RLev

Lev is GLev with $n = 1$. RLev is GLev with $k = 1$.

B-6 GGSW

The GGSW cryptosystem is a list of GLev ciphertexts. In the GGSW cryptosystem, the secret key S is a list of k polynomials (i.e., $S_0, S_1, \ldots S_{k-1}$), and each i-th GLev ciphertext in the GGSW ciphertext encrypts the plaintext $S_0 \cdot M, S_1 \cdot M, \ldots S_{k-1} \cdot M$, and M. This is visually depicted in ??.

B-6.1 Encryption

⟨Summary ??⟩ GGSW Encryption

$$\mathsf{GGSW}^{\beta,l}_{S,\sigma}(M) = \left\{ \{\mathsf{GLev}^{\beta,l}_{S,\sigma}(-S_i \cdot M)\}_{i=0}^{k-1}, \mathsf{GLev}^{\beta,l}_{S,\sigma}(M) \right\} \in \mathcal{R}^{(k+1)\cdot l\cdot(k+1)}_{\langle n,q\rangle}$$

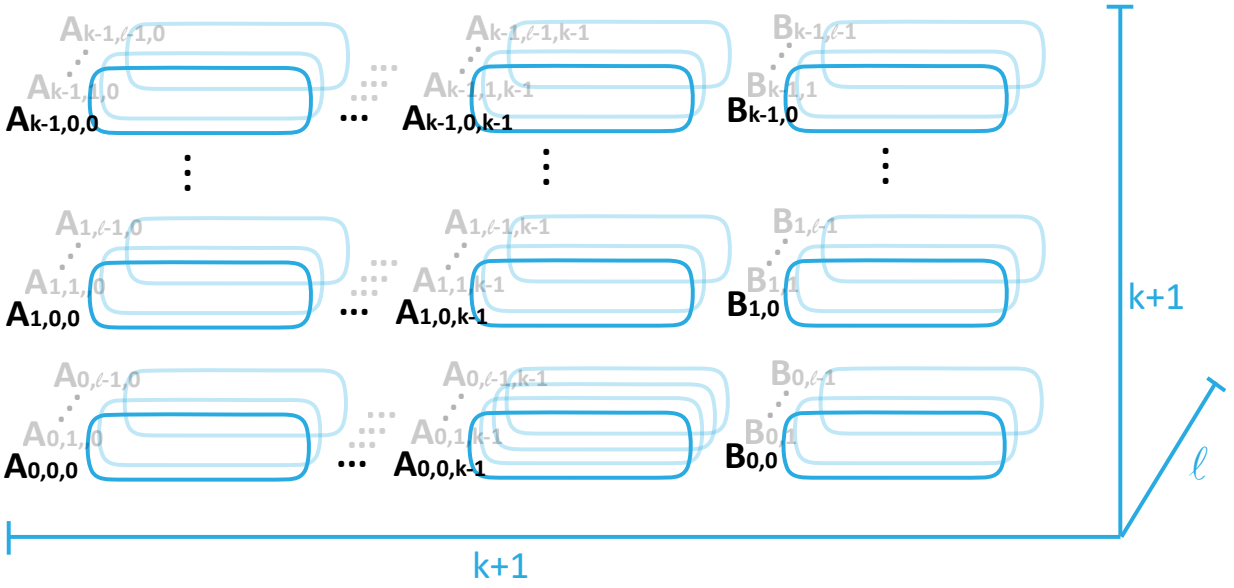


Figure 10: An illustration of a GGSW ciphertext

B-6.2 Decryption

It is sufficient to decrypt the GGSW ciphertext's any one GLev ciphertext's any one GLWE ciphertext, by using the secret S.

B-6.3 GSW and RGSW

GSW is GLev with $n = 1$. RGSW is GLev with $k = 1$.

Part III
Generic Fully Homomorphic Encryption

This chapter explains the generic techniques of homomorphic computation adopted by various FHE schemes such as TFHE, CKKS, BGV, and BFV,

As we learned from **??**, $\mathsf{GLWE}_{S,\sigma}(\Delta M) = (A_0, A_1, \cdots, A_{k-1}, B) \in \mathcal{R}^{k+1}_{\langle n,q\rangle}$, where $\mathcal{R}_{\langle n,q\rangle} = \mathbb{Z}_q[x]/(x^n+1)$, and B is computed as $B = \sum_{i=0}^{k-1}(A_i \cdot S_i) + \Delta \cdot M + E$. Each A_i is an $(n-1)$-degree polynomial as a public key, whose each coefficient is uniformly randomly sampled from $\mathcal{R}_{\langle n,q\rangle}$. E is an $(n-1)$-degree polynomial as a noise, whose each coefficient is sampled from $\mathcal{R}_{\langle n,q\rangle}$ based on the Gaussian distribution χ_σ. S is a list of k $(n-1)$-degree polynomials as a secret key, such that $S = (S_0, S_1, \cdots S_{k-1}) \in \mathcal{R}^k_{\langle n,q\rangle}$, and each polynomial S_i's each coefficient is a randomly sampled binary number in $\mathbb{Z}_2$ (i.e., $\{0,1\}$).

Based on this GLWE setup, this section will explain the following 5 homomorphic operations: ciphertext-to-ciphertext addition, ciphertext-to-plaintext addition, ciphertext-to-plaintext multiplication, ciphertext-to-ciphertext multiplication, and key switching.

C-1 GLWE Ciphertext-to-Ciphertext Addition

- Reference: TFHE Deep Dive - Part II - Encodings and linear leveled operations [**?**]

Suppose we have two GLWE ciphertexts encrypting two different plaintexts $M^{\langle 1 \rangle}, M^{\langle 2 \rangle}$:
$\mathsf{GLWE}_{S,\sigma}(\Delta M^{\langle 1 \rangle}) = C^{\langle 1 \rangle} = (A_0^{\langle 1 \rangle}, A_1^{\langle 1 \rangle}, \dots A_{k-1}^{\langle 1 \rangle}, B^{\langle 1 \rangle}) \in \mathcal{R}_{\langle n,q \rangle}^{k+1}$
$\mathsf{GLWE}_{S,\sigma}(\Delta M^{\langle 2 \rangle}) = C^{\langle 2 \rangle} = (A_0^{\langle 2 \rangle}, A_1^{\langle 2 \rangle}, \dots A_{k-1}^{\langle 2 \rangle}, B^{\langle 2 \rangle}) \in \mathcal{R}_{\langle n,q \rangle}^{k+1}$

Let's define the following TFHE ciphertext addition operation:
$C^{\langle 1 \rangle} + C^{\langle 2 \rangle} = (A_0^{\langle 1 \rangle} + A_0^{\langle 2 \rangle},\ A_1^{\langle 1 \rangle} + A_1^{\langle 2 \rangle},\ \dots A_{k-1}^{\langle 1 \rangle} + A_{k-1}^{\langle 2 \rangle},\ B^{\langle 1 \rangle} + B^{\langle 2 \rangle})$

Then, the following is true:

⟨Summary ??⟩ GLWE Homomorphic Addition

$$\begin{aligned}
&\mathsf{GLWE}_{S,\sigma}(\Delta M^{\langle 1 \rangle}) + \mathsf{GLWE}_{S,\sigma}(\Delta M^{\langle 2 \rangle}) \\
&= (\{A_i^{\langle 1 \rangle}\}_{i=0}^{k-1},\ B^{\langle 1 \rangle}) + (\{A_i^{\langle 2 \rangle}\}_{i=0}^{k-1},\ B^{\langle 2 \rangle}) \\
&= (\{A_i^{\langle 1 \rangle} + A_i^{\langle 2 \rangle}\}_{i=0}^{k-1},\ B^{\langle 1 \rangle} + B^{\langle 2 \rangle}) \\
&= \mathsf{GLWE}_{S,\sigma}(\Delta(M^{\langle 1 \rangle} + M^{\langle 2 \rangle}))
\end{aligned}$$

This means that adding two TFHE ciphertexts and decrypting the resulting ciphertext gives the same effect as adding two original (Δ-scaled) plaintexts: $\Delta M^{\langle 1 \rangle} + \Delta M^{\langle 2 \rangle} = \Delta \cdot (M^{\langle 1 \rangle} + M^{\langle 2 \rangle})$.

Proof.

1. Define the following notations:
   $A_0^{\langle 3 \rangle} = A_0^{\langle 1 \rangle} + A_0^{\langle 2 \rangle}$
   $A_1^{\langle 3 \rangle} = A_1^{\langle 1 \rangle} + A_1^{\langle 2 \rangle}$
   ...
   $A_{k-1}^{\langle 3 \rangle} = A_{k-1}^{\langle 1 \rangle} + A_{k-1}^{\langle 2 \rangle}$
   $E^{\langle 3 \rangle} = E^{\langle 1 \rangle} + E^{\langle 2 \rangle}$
   $B^{\langle 3 \rangle} = B^{\langle 1 \rangle} + B^{\langle 2 \rangle}$
2. Derive the following:
   $$\begin{aligned}
   B^{\langle 3 \rangle} &= B^{\langle 1 \rangle} + B^{\langle 2 \rangle} \\
   &= \sum_{i=0}^{k-1} (A_i^{\langle 1 \rangle} \cdot S_i) + \Delta \cdot M^{\langle 1 \rangle} + E^{\langle 1 \rangle} + \sum_{i=0}^{k-1} (A_i^{\langle 2 \rangle} \cdot S_i) + \Delta \cdot M^{\langle 2 \rangle} + E^{\langle 2 \rangle} \\
   &= \sum_{i=0}^{k-1} ((A_i^{\langle 1 \rangle} + A_i^{\langle 2 \rangle}) \cdot S_i) + \Delta \cdot (M^{\langle 1 \rangle} + M^{\langle 2 \rangle}) + (E^{\langle 1 \rangle} + E^{\langle 2 \rangle}) \ \text{\# commutative and distributive rules} \\
   &= \sum_{i=0}^{k-1} (A_i^{\langle 3 \rangle} \cdot S_i) + \Delta \cdot (M^{\langle 1 \rangle} + M^{\langle 2 \rangle}) + E^{\langle 3 \rangle}
   \end{aligned}$$
3. Since $B^{\langle 3 \rangle} = \sum_{i=0}^{k-1} (A_i^{\langle 3 \rangle} \cdot S_i) + \Delta \cdot (M^{\langle 1 \rangle} + M^{\langle 2 \rangle}) + E^{\langle 3 \rangle}$,

this means that $(A_0^{\langle 3\rangle}, A_1^{\langle 3\rangle}, A_2^{\langle 3\rangle}, \ldots A_{k-1}^{\langle 3\rangle}, B^{\langle 3\rangle})$ form the ciphertext: $\mathsf{GLWE}_{S,\sigma}(\Delta \cdot (M^{\langle 1\rangle} + M^{\langle 2\rangle}))$.

4. Thus,
$= \mathsf{GLWE}_{S,\sigma}(\Delta M^{\langle 1\rangle}) + \mathsf{GLWE}_{S,\sigma}(\Delta M^{\langle 2\rangle})$
$= (A_0^{\langle 1\rangle} + A_0^{\langle 2\rangle},\ A_1^{\langle 1\rangle} + A_1^{\langle 2\rangle},\ \ldots A_{k-1}^{\langle 1\rangle} + A_{k-1}^{\langle 2\rangle},\ B^{\langle 1\rangle} + B^{\langle 2\rangle})$
$= (A_0^{\langle 3\rangle}, A_1^{\langle 3\rangle}, A_2^{\langle 3\rangle}, \ldots A_{k-1}^{\langle 3\rangle}, B^{\langle 3\rangle})$
$= (\{A_i^{\langle 3\rangle}\}_{i=0}^{k-1}, B^{\langle 3\rangle})$
$= \mathsf{GLWE}_{S,\sigma}(\Delta(M^{\langle 1\rangle} + M^{\langle 2\rangle}))$

□

C-1.1 Discussion

Noise Elimination: If we decrypt $\mathsf{GLWE}_{S,\sigma}(\Delta(M^{\langle 1\rangle} + M^{\langle 2\rangle}))$ by using the secret key S, then we get the plaintext $M^{\langle 1\rangle} + M^{\langle 2\rangle}$. Meanwhile, $A_1^{\langle 3\rangle}, A_2^{\langle 3\rangle}, \ldots A_{k-1}^{\langle 3\rangle}, E^{\langle 3\rangle}$ get eliminated by rounding after decryption, regardless of whatever their randomly sampled values were during encryption.

Noise Growth: Note that after decryption, the original ciphertext C's noise has increased from $E^{\langle 1\rangle}$ and $E^{\langle 2\rangle}$ to $E^{\langle 3\rangle} = E^{\langle 1\rangle} + E^{\langle 2\rangle}$. However, if the noise is sampled from a Gaussian distribution with the mean $\mu = 0$, then the addition of multiple noises will converge to 0. Therefore, there is not much issue of noise growth in the homomorphic addition of two ciphertexts.

Hard Threshold on the Plaintext's Value Without Modulo Reduction t: During homomorphic operations (e.g., addition or multiplication) and decryption, the AS and B terms in the $B = AS + \Delta M + E + kq$ relation are allowed to wrap around modulo q indefinitely, because regardless of whatever their wrapping count is, the final decryption step will always subtract B by AS, outputting $\Delta M + E + k'q = \Delta M + E \pmod q$, and the $k'q$ term is always exactly eliminated by modulo reduction by q. After that, we can correctly recover M by computing $\left\lceil \frac{\Delta M + E \bmod q}{\Delta} \right\rfloor$, eliminating the noise E. However, as we explained in Summary **??** in **??**), if the error bound $\frac{kt + e}{\lfloor \frac{q}{t} \rfloor} < \frac{1}{2}$ breaks (where e can be any coefficient of E), then modulo reduction by q starts to contaminate the scaled plaintext bits. This violation of the error bound occurs when the noise e grows too much over homomorphic operations, or the ciphertext modulus q is not sufficiently larger than the plaintext modulus t. If $q \gg t$, the scheme can take on a big kt value (i.e., the plaintext value can wrap around the plaintext modulus t many times across its homomorphic operations). The error bound constraint $\frac{kt + e}{\lfloor \frac{q}{t} \rfloor} < \frac{1}{2}$ is used in the BFV scheme.

C-2 GLWE Ciphertext-to-Plaintext Addition

Suppose we have a GLWE ciphertext C and a new plaintext polynomial Λ as follows:
$C = \mathsf{GLWE}_{S,\sigma}(\Delta M) = (A_0, A_1, \cdots, A_{k-1}, B) \in \mathcal{R}^{k+1}_{\langle n,q \rangle}$
Λ: a new plaintext polynomial
$\Delta\Lambda$: a Δ-scaled new plaintext polynomial

Let's define the following TFHE ciphertext-to-plaintext addition operation:
$C + \Delta\Lambda = (A_0,\ A_1,\ \cdots, A_{k-1},\ B + \Delta\Lambda)$

Then, the following is true:

⟨Summary ??⟩ GLWE Homomorphic Addition with a Plaintext

$$\begin{aligned}
&\mathsf{GLWE}_{S,\sigma}(\Delta M) + \Delta\Lambda \\
&= (\{A_i^{\langle 1 \rangle}\}_{i=0}^{k-1},\ B^{\langle 1 \rangle}) + \Delta\Lambda \\
&= (\{A_i^{\langle 1 \rangle}\}_{i=0}^{k-1},\ B^{\langle 1 \rangle} + \Delta\Lambda) \\
&= \mathsf{GLWE}_{S,\sigma}(\Delta(M + \Lambda))
\end{aligned}$$

This means that a plaintext polynomial to a TFHE ciphertext and decrypting it gives the same result as adding two original plaintexts.

Proof.

1. Since $B = \sum_{i=0}^{k-1} (A_i \cdot S_i) + \Delta \cdot M + E$,
   $B + \Delta \cdot \Lambda = \sum_{i=0}^{k-1} (A_i \cdot S_i) + \Delta \cdot M + E + \Delta \cdot \Lambda = \sum_{i=0}^{k-1} (A_i \cdot S_i) + \Delta \cdot (M + \Lambda) + E$
   This means that $(A_0, A_1, \ldots A_{k-1}, B + \Delta\Lambda)$ form the ciphertext $\mathsf{GLWE}_{S,\sigma}(\Delta(\Lambda + M))$
2. Thus,
   $\mathsf{GLWE}_{S,\sigma}(\Delta M) + \Delta\Lambda$
   $= (A_0,\ A_1, \ldots A_{k-1},\ B + \Delta\Lambda)$
   $= (\{A_i^{\langle 1 \rangle}\}_{i=0}^{k-1},\ B + \Delta\Lambda)$
   $= \mathsf{GLWE}_{S,\sigma}(\Delta(M + \Lambda))$

□

C-2.1 Discussion

Noise Growth: Note that after decryption, the original ciphertext $C + \Lambda$'s noise E stays the same as before. This means that ciphertext-to-plaintext multiplication does not increase the noise level.

C-3 GLWE Ciphertext-to-Plaintext Multiplication

- Reference: TFHE Deep Dive - Part II - Encodings and linear leveled operations [**?**]

Suppose we have a GLWE ciphertext C:
$C = \mathsf{GLWE}_{S,\sigma}(\Delta M) = (A_0, A_1, ... A_{k-1}, B) \in \mathcal{R}^{k+1}_{\langle n,q \rangle}$

and a new plaintext polynomial Λ as follows:
$\Lambda = \sum_{i=0}^{n-1} (\Lambda_i \cdot X_i) \in \mathcal{R}_{\langle n,q \rangle}$

Let's define the following ciphertext-to-plaintext multiplication operation:
$\Lambda \cdot C = (\Lambda \cdot A_0, \Lambda \cdot A_1, ... \Lambda \cdot A_{k-1}, \Lambda \cdot B)$

We assume that we always do polynomial-to-polynomial multiplications efficiently in $O(n \log n)$ by using the NTT technique (**??**). Then, the following is true:

⟨Summary ??⟩ GLWE Ciphertext-to-Plaintext Multiplication

$$
\begin{aligned}
&\Lambda \cdot \mathsf{GLWE}_{S,\sigma}(\Delta M) \\
&= \Lambda \cdot (\{A_i^{\langle 1 \rangle}\}_{i=0}^{k-1},\ B^{\langle 1 \rangle}) \\
&= (\{\Lambda \cdot A_i^{\langle 1 \rangle}\}_{i=0}^{k-1},\ \Lambda \cdot B^{\langle 1 \rangle}) \\
&= \mathsf{GLWE}_{S,\sigma}(\Delta(M \cdot \Lambda))
\end{aligned}
$$

This means that multiplying a plaintext (polynomial) by a ciphertext (polynomial) and decrypting it gives the same result as multiplying the two original plaintext polynomials.

Proof.

1. Define the following notations:
   $A'_0 = \Lambda \cdot A_0$
   $A'_1 = \Lambda \cdot A_1$
   ...
   $A'_{k-1} = \Lambda \cdot A_{k-1}$
   $E' = \Lambda \cdot E$
   $B' = \Lambda \cdot B$

2. Derive the following:
   $B' = \Lambda \cdot B$
   $= \Lambda \cdot (\sum_{i=0}^{k-1} (A_i \cdot S_i) + \Delta \cdot M + E) = \sum_{i=0}^{k-1} (\Lambda \cdot A_i \cdot S_i) + \Delta \cdot \Lambda \cdot M + \Lambda \cdot E$
   # by the distributive property of a polynomial ring
   $= \sum_{i=0}^{k-1} ((\Lambda \cdot A_i) \cdot S_i) + \Delta \cdot (\Lambda \cdot M) + (\Lambda \cdot E)$
   $= \sum_{i=0}^{k-1} (A'_i \cdot S_i) + \Delta \cdot (\Lambda \cdot M) + (E')$

3. Since $B' = \sum_{i=0}^{k-1} (A'_i \cdot S_i) + \Delta \cdot (\Lambda \cdot M) + (E')$,
   $(A'_0, A'_1, A'_2, \ldots A'_{k-1}, B')$ form the ciphertext $\mathsf{GLWE}_{S,\sigma}(\Delta \cdot \Lambda \cdot M)$.
4. Thus,
   $\Lambda \cdot \mathsf{GLWE}_{S,\sigma}(\Delta M)$
   $= (\Lambda \cdot A_0,\ \Lambda \cdot A_1, \ldots \Lambda \cdot A_{k-1},\ \Lambda \cdot B)$
   $= (\{A'_i\}_{i=0}^{k-1},\ \Lambda \cdot B)$
   $= \mathsf{GLWE}_{S,\sigma}(\Delta(M \cdot \Lambda))$

□

If we decrypt $\mathsf{GLWE}_{S,\sigma}(\Delta \cdot \Lambda \cdot M)$ by using S, then we get the plaintext $\Lambda \cdot M$. Meanwhile, $A'_0, A'_1, A'_2, \ldots A'_{k-1}, E'$ get eliminated by rounding during decryption, regardless of whatever their values were randomly sampled during encryption.

The noise is a bigger problem now, because after decryption, the original ciphertext C's noise has increased from E to $E' = \Lambda \cdot E$. This means that if we continue multiplication computations without decrypting the ciphertext to blow away the noise E', it will continue growing more and eventually the noise in the lower bit area in B will overflow to the scaled plaintext bit area. If this happens, the noise E' won't be blown away during decryption, ending up corrupting the plaintext M. Therefore, if the constant Λ is big, it is recommended to use gadget decomposition (**??**), which we will explain in the next subsection.

C-3.1 Gadget Decomposition for Noise Suppression

In ciphertext-to-plaintext multiplication $\Lambda \cdot \mathsf{GLWE}_{S,\sigma}(M)$, the noise E grows to $E' = \Lambda \cdot E$. To limit this noise growth, we introduce a technique based on decomposing Λ (**??**) and a GLev encryption (**??**) of M as follows:

$$\Lambda = \Lambda_1 \frac{q}{\beta^1} + \Lambda_2 \frac{q}{\beta^2} + \cdots + \Lambda_l \frac{q}{\beta^l} \longrightarrow \mathsf{Decomp}^{\beta,l}(\Lambda) = (\Lambda_1, \Lambda_2, \cdots, \Lambda_l)$$

$$\mathsf{GLev}^{\beta,l}_{S,\sigma}(\Delta M) = \left\{ \mathsf{GLWE}_{S,\sigma}\left(\Delta M \frac{q}{\beta^1}\right), \mathsf{GLWE}_{S,\sigma}\left(\Delta M \frac{q}{\beta^2}\right), \cdots \mathsf{GLWE}_{S,\sigma}\left(\Delta M \frac{q}{\beta^l}\right) \right\}$$

We will encrypt the plaintext M as $\mathsf{GLev}^{\beta,l}_{S,\sigma}(\Delta M)$ instead of $\mathsf{GLWE}_{S,\sigma}(\Delta M)$, and compute $\mathsf{Decomp}(\mathsf{Decomp}^{\beta,l}(\Lambda) \cdot \mathsf{GLev}^{\beta,l}_{S,\sigma}(\Delta M)$ instead of $\Lambda \cdot \mathsf{GLWE}_{S,\sigma}(\Delta M)$. Notice that the results of both computations are the same as follows:

$$\begin{aligned}
&\mathsf{Decomp}^{\beta,l}(\Lambda) \cdot \mathsf{GLev}^{\beta,l}_{S,\sigma}(\Delta M) \\
&= (\Lambda_1, \Lambda_2, \cdots, \Lambda_l) \cdot \left(\mathsf{GLWE}_{S,\sigma}\left(\frac{q}{\beta}\Delta M\right),\ \mathsf{GLWE}_{S,\sigma}\left(\frac{q}{\beta^2}\Delta M\right),\ \cdots,\ \mathsf{GLWE}_{S,\sigma}\left(\frac{q}{\beta^l}\Delta M\right)\right) \\
&= \Lambda_1 \cdot \mathsf{GLWE}_{S,\sigma}\left(\frac{q}{\beta}\Delta M\right) + \Lambda_2 \cdot \mathsf{GLWE}_{S,\sigma}\left(\frac{q}{\beta^2}\Delta M\right) + \cdots + \Lambda_l \cdot \mathsf{GLWE}_{S,\sigma}\left(\frac{q}{\beta^l}\Delta M\right) \\
&= \mathsf{GLWE}_{S,\sigma}\left(\Lambda_1 \cdot \frac{q}{\beta} M\right) + \mathsf{GLWE}_{S,\sigma}\left(\Lambda_2 \cdot \frac{q}{\beta^2} M\right) + \cdots + \mathsf{GLWE}_{S,\sigma}\left(\Lambda_l \cdot \frac{q}{\beta^l} M\right) \\
&= \mathsf{GLWE}_{S,\sigma}\left(\Lambda_1 \cdot \frac{q}{\beta}\Delta M + \Lambda_2 \cdot \frac{q}{\beta^2}\Delta M + \cdots + \Lambda_l \cdot \frac{q}{\beta^l}\Delta M\right) \\
&= \mathsf{GLWE}_{S,\sigma}\left(\left(\Lambda_1 \cdot \frac{q}{\beta} + \Lambda_2 \cdot \frac{q}{\beta^2} + \cdots + \Lambda_l \cdot \frac{q}{\beta^l}\right) \cdot \Delta M\right) \\
&= \mathsf{GLWE}_{S,\sigma}\left(\Lambda \cdot \Delta M\right) \\
&= \Lambda \cdot \mathsf{GLWE}_{S,\sigma}(\Delta M)
\end{aligned}$$

While the computation results are the same, as we decompose Λ into smaller plaintext polynomials $\Lambda_1, \Lambda_2, \cdots, \Lambda_l$, the generated noise by each of l plaintext-to-ciphertext multiplications becomes smaller. Given the noise of each GLWE ciphertext in the GLev ciphertext is E_i, the final noise of the ciphertext-to-plaintext multiplication is $\sum_{i=1}^{l} \Lambda_i \cdot E_i$, which is much smaller than $\Lambda \cdot E$ (because the coefficients of each decomposed polynomial Λ_i are significantly smaller than those of Λ). This is visually depicted in **??**.

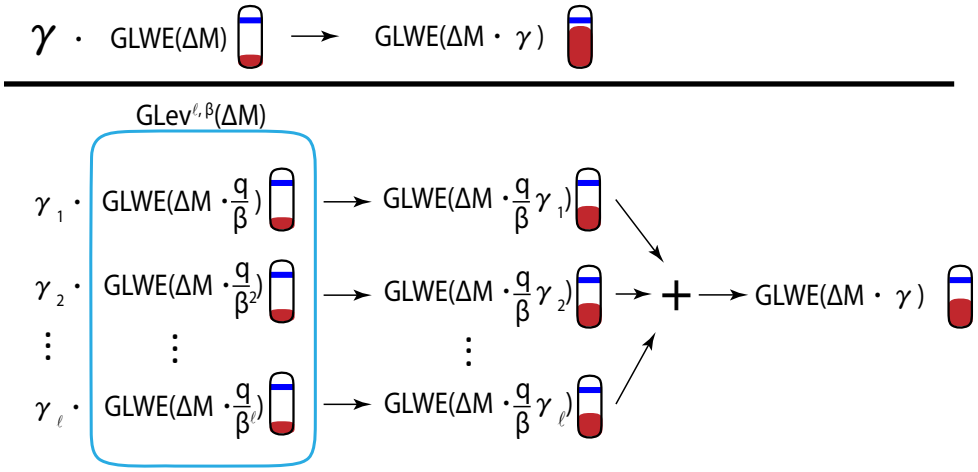


Figure 11: Noise reduction in ciphertext-to-plaintext multiplication by gadget decomposition.

However, we cannot use this decomposition technique to the resulting ciphertext again, because the output of this algorithm is a GLWE ciphertext and converting it into a GLev ciphertext without decrypting it costs much noise and computation time (as we need to multiply the GLWE ciphertext by $\frac{q}{\beta}, \frac{q}{\beta^2}, \cdots, \frac{q}{\beta^l}$).

As a more efficient technique to re-initialize the noise E, we will describe TFHE's noise bootstrapping technique in **??**.

C-4 GLWE Modulus Switching

In the GLWE cryptosystem, modulus switching is a process of changing a ciphertext's modulo domain to a smaller (or larger) one, while ensuring that the ciphertext still decrypts to the same plaintext. For example, suppose we have the ciphertext $\mathsf{LWE}_{S,\sigma}(\Delta m) \in \mathbb{Z}_q^{k+1}$. If we switch the ciphertext's modulo from $q \to q'$, then the ciphertext is converted into $\mathsf{LWE}_{S,\sigma}\left(\Delta \frac{q'}{q} m\right) \in \mathbb{Z}_{q'}^{k+1}$. The ciphertext's all other components such as the noise (e) and public keys ($a_0, a_1, \cdots, a_{k-1}, b$) are scaled by $\frac{q'}{q}$, becoming $\left\lceil e\frac{q'}{q} \right\rfloor$, $\left(\left\lceil a_0\frac{q'}{q} \right\rfloor, \left\lceil a_1\frac{q'}{q} \right\rfloor, \cdots, \left\lceil a_{k-1}\frac{q'}{q} \right\rfloor, \left\lceil b\frac{q'}{q} \right\rfloor\right)$. To switch the modulo of an LWE ciphertext, we use the modulo rescaling technique learned from **??**. The same modulus switching technique can also be applied to RLWE ciphertexts. In this section, we will show how to switch (i.e., rescale) the modulo of LWE and RLWE ciphertexts and prove its correctness.

C-4.1 LWE Modulus Switching

- Reference: Modulus Switching in LWE

Recall that the LWE cryptosystem (**??**) comprises the following components:

- **Setup:** $\Delta = \frac{q}{t}, \quad S = (s_0, s_1, \ \cdots \ s_{k-1}) \xleftarrow{\$} \mathbb{Z}_2^k$
- **Encryption Input:** $m \in \mathbb{Z}_t, \quad A = (a_0, a_1, \ \cdots \ a_{k-1}) \xleftarrow{\$} \mathbb{Z}_q^k, \quad e \xleftarrow{\chi_\sigma} \mathbb{Z}_q$
- **Encryption:** $\mathsf{LWE}_{S,\sigma}(\Delta m) = (A, b) \in \mathbb{Z}_q^{k+1} \quad$ (where $b = A \cdot S + \Delta m + e \in \mathbb{Z}_q$)
- **Decryption:** $\mathsf{LWE}_{S,\sigma}^{-1}(C) = b - A \cdot S = \Delta m + e \quad \in \mathbb{Z}_q$

In the LWE cryptosystem, modulus switching is a process of converting an original LWE ciphertext's modulo domain to a smaller modulo domain. This can be seen as scaling down all components, except for the plaintext m and the secret key S, in the original LWE ciphertext to a smaller domain. This operation preserves the size and integrity of the original plaintext m, while the scaling factor Δ gets reduced to a smaller value $\hat{\Delta}$ and the noise e to a smaller (reduced) noise $\hat{e}$ (note that noise alteration does not affect the original plaintext m, because the noise gets rounded away after decryption, anyway), and A also gets scaled down to a smaller $\hat{A}$. Modulus switching is used for computational efficiency during TFHE's bootstrapping (which will be discussed in **??**). Modulus switching is also used for implementing the ciphertext-to-ciphertext multiplication algorithm in BGV (**??**) and CKKS (**??**).

The high-level idea of LWE modulus switch is to rescale the congruence relationship of the LWE scheme. LWE's homomorphic computation algorithms include the following: ciphertext-to-ciphertext addition, ciphertext-to-plaintext addition, ciphertext-to-plaintext multiplication, ciphertext-to-ciphertext multiplication. However, all congruence relationships used in these algorithms are essentially rewritten versions of the following single fundamental congruence relationship: $b = A \cdot S + \Delta m + e \bmod q$. Thus, modulus switch of an LWE ciphertext from $q \to q'$ is equivalent to rescaling the modulo of the above congruence relationship from $q \to q'$.

Based on this insight, the LWE cryptosystem's modulus switching from $q \to \hat{q}$ (where $q > \hat{q}$) is a process of converting the original LWE ciphertext $\mathsf{LWE}_{S,\sigma}(\Delta m)$ as follows:

⟨Summary ??⟩ LWE Modulus Switching

Given an LWE ciphertext (A, b) where $b = A \cdot S + \Delta m + e \bmod q$ and $m \in \mathbb{Z}_t$, modulus switch of the ciphertext from q to $\hat{q}$ is equivalent to updating (A, b) to $(\hat{A}, \hat{b})$ as follows:

$\hat{A} = (\hat{a}_0, \hat{a}_1, \cdots \hat{a}_{k-1})$, where each $\hat{a}_i = \left\lceil a\frac{\hat{q}}{q} \right\rfloor \in \mathbb{Z}_{\hat{q}}$ # $\lceil \rfloor$ means rounding to the nearest integer

$\hat{b} = \left\lceil b\frac{\hat{q}}{q} \right\rfloor \in \mathbb{Z}_{\hat{q}}$

$\mathsf{LWE}_{S,\sigma}(\hat{\Delta} m) = (\hat{a}_0, \hat{a}_1, \cdots, \hat{b}) \in \mathbb{Z}_{\hat{q}}^{k+1}$

The above update effectively changes $\hat{e}$ and $\hat{\Delta}$ as follows:

$\hat{e} = \left\lceil e\frac{\hat{q}}{q} \right\rfloor \in \mathbb{Z}_{\hat{q}}$,

$\hat{\Delta} = \Delta\frac{\hat{q}}{q}$ # which should be an integer

Meanwhile, S and m stay the same as before.

Note that in order for $(\hat{a}_0, \hat{a}_1, \cdots, \hat{b}) \in \mathbb{Z}_{\hat{q}}^{k+1}$ to be a valid LWE ciphertext of $\hat{\Delta} m$, we need to prove that the following relationship holds:

$$\hat{b} = \sum_{i=0}^{k-1} \hat{a}_i \cdot s_i + \hat{\Delta} m + \hat{e} \in \mathbb{Z}_{\hat{q}}$$

Proof.

1. Note the following:
   - $\hat{b} = \left\lceil b\frac{\hat{q}}{q} \right\rfloor = b\frac{\hat{q}}{q} + \epsilon_b$ (where $-0.5 < \epsilon_b < 0.5$, a rounding drift error)
   - $\hat{a_i} = \left\lceil a_i\frac{\hat{q}}{q} \right\rfloor = a_i\frac{\hat{q}}{q} + \epsilon_{a_i}$ (where $-0.5 < \epsilon_{a_i} < 0.5$)
   - $\hat{e} = \left\lceil e\frac{\hat{q}}{q} \right\rfloor = e\frac{\hat{q}}{q} + \epsilon_e$ (where $-0.5 < \epsilon_e < 0.5$)
2. Note the following:
   - $b = \vec{a} \cdot \vec{s} + \Delta m + e = \sum_{i=0}^{k-1} (a_i s_i) + \Delta m + e \in \mathbb{Z}_q$
   - $b = \sum_{i=0}^{k-1} (a_i s_i) + \Delta m + e + H \cdot q$ (where modulo q is replaced by adding $H \cdot q$, some multiple of q)
3. According to step 1 and 2:

$$\begin{aligned}
&\hat{b} = b\frac{\hat{q}}{q} + \epsilon_b \in \mathbb{Z}_{\hat{q}} \\
&= \left(\sum_{i=0}^{k-1} (a_i s_i) + \Delta m + e + H \cdot q \right) \cdot \frac{\hat{q}}{q} + \epsilon_b \\
&= \frac{\hat{q}}{q} \cdot \sum_{i=0}^{k-1} (a_i s_i) + \frac{\hat{q}}{q} \cdot \Delta m + \frac{\hat{q}}{q} \cdot e + \frac{\hat{q}}{q} \cdot H \cdot q + \epsilon_b \\
&= \sum_{i=0}^{k-1} \left(\frac{\hat{q}}{q} \cdot a_i s_i \right) + \hat{\Delta} m + (\hat{e} - \epsilon_e) + \hat{q} \cdot H + \epsilon_b
\end{aligned}$$

$$= \sum_{i=0}^{k-1} ((\hat{a}_i - \epsilon_{a_i}) \cdot s_i) + \hat{\Delta}m + \hat{e} - \epsilon_e + \hat{q} \cdot H + \epsilon_b$$
$$= \sum_{i=0}^{k-1} (\hat{a}_i s_i - \epsilon_{a_i} s_i) + \hat{\Delta}m + \hat{e} - \epsilon_e + \epsilon_b \in \mathbb{Z}_{\hat{q}}$$
$$= \sum_{i=0}^{k-1} \hat{a}_i s_i + \hat{\Delta}m + \left(\hat{e} - \epsilon_e + \epsilon_b - \sum_{i=0}^{k-1} \epsilon_{a_i} s_i \right) \in \mathbb{Z}_{\hat{q}}$$
$$= \sum_{i=0}^{k-1} \hat{a}_i s_i + \hat{\Delta}m + \hat{e} + \epsilon_{all} \in \mathbb{Z}_{\hat{q}} \ \# \text{ where } \epsilon_{all} = \left(-\epsilon_e + \epsilon_b - \sum_{i=0}^{k-1} \epsilon_{a_i} s_i \right)$$

The biggest possible value for ϵ_{all} is,
$\epsilon_{all} = |-0.5| + |0.5| + |-0.5 \cdot k| = 1 + 0.5k$
So, LWE modulus switching results in an approximate congruence relationship (**??**). However, if $\hat{\Delta}$ is large enough, $\epsilon_{all} = 1 + 0.5k$ will be shifted to the right upon LWE decryption and get eliminated, and finally we can recover the original m. Also, in practice, the term $\sum_{i=0}^{k-1} \epsilon_{a_i} s_i$ would converge to 0 for a sufficiently large k, because each a_i is uniformly sampled and s_i is also uniformly sampled.

Caution: If $\hat{\Delta}$ is not large enough, then ϵ_{all} may not get eliminated during decryption and corrupt the plaintext m. Also, if $\Delta \to \hat{\Delta}$ shrinks too much, then the distance between $\hat{\Delta}m$ and $\hat{e}$ would become too narrow and the rounding process of $\hat{e} = \left\lceil e\frac{\hat{q}}{q} \right\rfloor$ may end up overlapping the least significant bit of $\hat{\Delta}m$, corrupting the plaintext.

4. To summarize, $\hat{b}$ is approximately as follows:
$\hat{b} = \sum_{i=0}^{k-1} \hat{a}_i s_i + \hat{\Delta}m + \hat{e} + \epsilon_{all} \approx \sum_{i=0}^{k-1} \hat{a}_i s_i + \hat{\Delta}m + \hat{e} \in \mathbb{Z}_{\hat{q}}$
Thus, $(\hat{a}_0, \hat{a}_1, \cdots, \hat{b}) = \mathsf{LWE}_{S,\sigma}(\hat{\Delta}m)$, decrypting which will give us m.

□

C-4.2 Example

Suppose we have the following LWE setup:

$t = 4$
$q = 64$
$n = 4$
$\Delta = \frac{q}{t} = 16$
$m = 1 \in \mathbb{Z}_t$
$S = (s_0, s_1, s_2, s_3) = (0, 1, 1, 0) \in \mathbb{Z}_b^4$
$A = (a_0, a_1, a_2, a_3) = (-25, 12, -3, 7) \in \mathbb{Z}_q^4$
$e = 1 \in \mathbb{Z}_q$
$b = a_0 s_0 + a_1 s_1 + a_2 s_2 + a_3 s_3 + \Delta m + e = 26 \in \mathbb{Z}_q$
$\mathsf{LWE}_{S,\sigma}(\Delta m) = C = (a_0, a_1, a_2, a_3, b) = (-25, 12, -3, 7, 26) \in \mathbb{Z}_q^{n+1}$

Now, suppose we want modulus switching from $q = 64$ to $\hat{q} = 32$, which gives:

$\hat{\Delta} = \Delta \cdot \frac{32}{64} = 8$
$\hat{e} = \left\lceil 1 \cdot \frac{32}{64} \right\rfloor = 1$
$\mathsf{LWE}_{S,\sigma}(\hat{\Delta}m) = \hat{C} = (\hat{a_0}, \hat{a_1}, \hat{a_2}, \hat{a_3}, \hat{b})$
$= \left(\left\lceil -25 \cdot \frac{32}{64} \right\rfloor, \left\lceil 12 \cdot \frac{32}{64} \right\rfloor, \left\lceil -3 \cdot \frac{32}{64} \right\rfloor, \left\lceil 7 \cdot \frac{32}{64} \right\rfloor, \left\lceil 26 \cdot \frac{32}{64} \right\rfloor \right)$
$= (-12, 6, -1, 4, 13) \in \mathbb{Z}_{\hat{q}}^{n+1}$

Now, verify if the following LWE constraint holds:
$\hat{b} = \hat{a}_0 s_0 + \hat{a}_1 s_1 + \hat{a}_2 s_2 + \hat{a}_3 s_3 + \hat{\Delta} m + \hat{e} \in \mathbb{Z}_{32}$
$13 = 0 + 6 - 1 + 0 + 8 \cdot 1 + 1 \in \mathbb{Z}_{32}$
$13 \approx 14 \in \mathbb{Z}_{32}$
We got this small difference of 1 due to the rounding drift error of
$\hat{a_0} = \lceil -12.5 \rfloor = -12$ and $\hat{a_3} = \lceil 3.5 \rfloor = 4$.

If we solve the LWE decryption formula:
$\hat{b} - (\hat{a}_0 s_0 + \hat{a}_1 s_1 + \hat{a}_2 s_2 + \hat{a}_3 s_3) = 13 - 4 = 9 = \hat{m} + \hat{e} \in \mathbb{Z}_{32}$

$m = \left\lceil \frac{9}{\hat{\Delta}} \right\rfloor = \left\lceil \frac{9}{8} \right\rfloor = 1$, which is correct.

C-4.3 Discussion

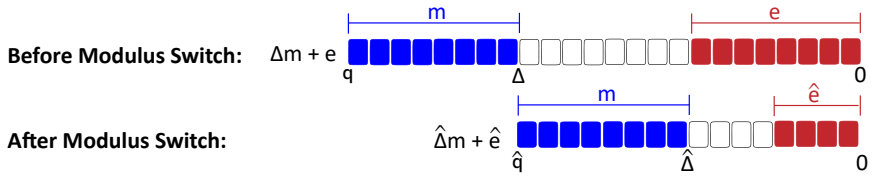


Figure 12: An illustration of scaled plaintext with a noise: $\Delta \cdot m + e \in \mathbb{Z}_q$

Reduced Distance between m and e: After modulus switching of an LWE ciphertext from $q \to \hat{q}$, the underlying plaintext (containing a noise) $\Delta m + e$ gets shrunk to $\hat{\Delta} m + \hat{e}$, as illustrated in **??**. Note that after the modulus switch from $q \to \hat{q}$, Δm is down-scaled to $\hat{\Delta} m$ without losing its bit data. Notably, the plaintext value m stays the same after the modulus switch, while its scaling factor Δ gets reduced to $\hat{\Delta}$ and the noise e gets reduced to $\hat{e}$. However, after the modulus switch, the distance between $\hat{e}$'s MSB and $\hat{\Delta} m$'s LSB gets reduced compared to the distance between e's MSB and Δm's LSB.

C-4.4 RLWE Modulus Switching

RLWE modulus switching is similar to LWE modulus switching. Recall that the RLWE cryptosystem (**??**) comprises the following components:

- **Setup:** $\Delta = \frac{q}{t}, \quad S = s_0 + s_1 X + s_2 X^2 + \cdots + s_{n-1} X^{n-1} \xleftarrow{\$} \mathcal{R}_{\langle n,2 \rangle}$

- **Encryption Input:**
  $M \in \mathcal{R}_{\langle n,t \rangle}$
  $A = a_0 + a_1 X + a_2 X^2 + \cdots + a_{n-1} X^{n-1} \xleftarrow{\$} \mathcal{R}_{\langle n,q \rangle}$
  $E = e_0 + e_1 X + e_2 X^2 + \cdots + e_{n-1} X^{n-1} \xleftarrow{\chi_\sigma} \mathcal{R}_{\langle n,q \rangle}$

- **Encryption:** $\mathsf{RLWE}_{S,\sigma}(\Delta \cdot M) = (A, B) \in \mathcal{R}^2_{\langle n,q \rangle}$
  , where $B = A \cdot S + \Delta \cdot M + E = b_0 + b_1 X + b_2 X^2 + \cdots + b_{n-1} X^{n-1}$

- **Decryption:** $\mathsf{RLWE}^{-1}_{S,\sigma}(C) = B - A \cdot S = \Delta M + E \ \in \mathcal{R}_{\langle n,q \rangle}$

RLWE modulus switching is done as follows:

⟨Summary ??⟩ RLWE Modulus Switching

For an RLWE ciphertext (A, B) where $B = AS + \Delta M + E$ and $M \in \mathcal{R}_{\langle n,q \rangle}$, modulus switch of the ciphertext from q to $\hat{q}$ is equivalent to updating (A, B) to $(\hat{A}, \hat{B})$ as follows:

$\hat{A} = \hat{a}_0 + \hat{a}_1 X + \hat{a}_2 X^2 + \cdots + \hat{a}_{n-1} X^{n-1}$, where each $\hat{a}_i = \left\lceil a_i \frac{\hat{q}}{q} \right\rfloor \in \mathbb{Z}_{\hat{q}}$

$\hat{B} = \hat{b}_0 + \hat{b}_1 X + \hat{b}_2 X^2 + \cdots + \hat{b}_{n-1} X^{n-1}$, where each $\hat{b}_i = \left\lceil b_i \frac{\hat{q}}{q} \right\rfloor \in \mathbb{Z}_{\hat{q}}$

$\mathsf{RLWE}_{S,\sigma}(\hat{\Delta} M) = (\hat{A}, \hat{B}) \in \mathcal{R}^2_{\langle n,\hat{q} \rangle}$

The above update effectively changes Δ and E as follows:

$\hat{\Delta} = \Delta \frac{\hat{q}}{q}$ # which should be an integer

$\hat{E} = \hat{e}_0 + \hat{e}_1 X + \hat{e}_2 X^2 + \cdots + \hat{e}_{n-1} X^{n-1}$, where each $\hat{e}_i = \left\lceil e_i \frac{\hat{q}}{q} \right\rfloor \in \mathbb{Z}_{\hat{q}}$

Meanwhile, S and M stay the same as before.

The proof is similar to that of LWE modulus switching.

Proof

1. Note the following:
   $\hat{b}_i = \left\lceil b_i \frac{\hat{q}}{q} \right\rfloor = b_i \frac{\hat{q}}{q} + \epsilon_{b_i}$ (where $-0.5 < \epsilon_{b_i} < 0.5$, a rounding drift error)
   $\hat{a}_i = \left\lceil a_i \frac{\hat{q}}{q} \right\rfloor = a_i \frac{\hat{q}}{q} + \epsilon_{a_i}$ (where $-0.5 < \epsilon_{a_i} < 0.5$)
   $\hat{e}_i = \left\lceil e_i \frac{\hat{q}}{q} \right\rfloor = e_i \frac{\hat{q}}{q} + \epsilon_{e_i}$ (where $-0.5 < \epsilon_{e_i} < 0.5$)
2. Note the following:
   $B - A \cdot S$
   $= (b_0 + b_1 X + \ \cdots \ + b_{n-1} X^{n-1}) - (a_0 + a_1 X + \ \cdots \ + a_{n-1} X^{n-1})(s_0 + s_1 X + \ \cdots \ + s_{n-1} X^{n-1})$
   $= \left(b_0 - \left(\sum_{i=0}^{0} (a_{0-i} s_i) - \sum_{i=1}^{n-1} (a_{n-i} s_i) \right) \right)$
   $+ \left(b_1 - \left(\sum_{i=0}^{1} (a_{1-i} s_i) - \sum_{i=2}^{n-1} (a_{n+1-i} s_i) \right) \right) \cdot X$

$$+\left(b_2-\left(\sum_{i=0}^{2}(a_{2-i}s_i)-\sum_{i=3}^{n-1}(a_{n+2-i}s_i)\right)\right)\cdot X^2$$

$$\vdots$$

$$+\left(b_{n-1}-\left(\sum_{i=0}^{n-1}(a_{n-1-i}s_i)-\sum_{i=n}^{n-1}(a_{n+n-1-i}s_i)\right)\right)\cdot X^{n-1} \text{ \# Grouping the terms by same exponents}$$

$$=\sum_{h=0}^{n-1}\left(b_h-\left(\sum_{i=0}^{h}(a_{h-i}s_i)-\sum_{i=h+1}^{n-1}(a_{n+h-i}s_i)\right)\right)\cdot X^h$$

Thus,

$$B=\sum_{h=0}^{n-1}b_hX^h$$

$$A\cdot S=\sum_{h=0}^{n-1}\left(\sum_{i=0}^{h}(a_{h-i}s_i)-\sum_{i=h+1}^{n-1}(a_{n+h-i}s_i)\right)\cdot X^h$$

3. Based on step 2,
$B=A\cdot S+\Delta M+E$
$\sum_{h=0}^{n-1}b_hX^h=\sum_{h=0}^{n-1}\left(\sum_{i=0}^{h}(a_{h-i}s_i)-\sum_{i=h+1}^{n-1}(a_{n+h-i}s_i)\right)\cdot X^h+\Delta\sum_{h=0}^{n-1}m_hX^h+\sum_{h=0}^{n-1}e_hX^h\in\mathbb{Z}_q$
$\sum_{h=0}^{n-1}b_hX^h=\sum_{h=0}^{n-1}\left(\sum_{i=0}^{h}(a_{h-i}s_i)-\sum_{i=h+1}^{n-1}(a_{n+h-i}s_i)\right)\cdot X^h+\Delta\sum_{h=0}^{n-1}m_hX^h+\sum_{h=0}^{n-1}e_hX^h+H\cdot q$
(where modulo q is replaced by adding $H\cdot q$, an $(n-1)$-degree polynomial whose each coefficient c_i is some multiple of q)

4. According to step 1 and 3, for each j in $0\le j\le n-1$:
$\hat{b}_j=b_j\frac{\hat{q}}{q}+\epsilon_{b_j}\in\mathbb{Z}_{\hat{q}}$
$=\left(\sum_{i=0}^{j}(a_{j-i}s_i)-\sum_{i=j+1}^{n-1}(a_{n+j-i}s_i)+\Delta m_j+e_j+c_j\cdot q\right)\cdot\frac{\hat{q}}{q}+\epsilon_{b_j}$
$=\frac{\hat{q}}{q}\sum_{i=0}^{j}(a_{j-i}s_i)-\frac{\hat{q}}{q}\sum_{i=j+1}^{n-1}(a_{n+j-i}s_i)+\frac{\hat{q}}{q}\cdot\Delta m_j+\frac{\hat{q}}{q}e_j+\frac{\hat{q}}{q}\cdot c_j\cdot q+\epsilon_{b_j}$
$=\sum_{i=0}^{j}\left(\frac{\hat{q}}{q}\cdot a_{j-i}s_i\right)-\sum_{i=j+1}^{n-1}\left(\frac{\hat{q}}{q}\cdot a_{n+j-i}s_i\right)+\hat{\Delta}m_j+(\hat{e}_j-\epsilon_{e_j})+\hat{q}\cdot c_j+\epsilon_{b_j}$
$=\sum_{i=0}^{j}((\hat{a}_{j-i}-\epsilon_{a_{j-i}})\cdot s_i)-\sum_{i=j+1}^{n-1}((\hat{a}_{n+j-i}-\epsilon_{a_{n+j-i}})\cdot s_i)+\hat{\Delta}m_j+(\hat{e}_j-\epsilon_{e_j})+\hat{q}\cdot c_j+\epsilon_{b_j}$
$=\sum_{i=0}^{j}(\hat{a}_{j-i}s_i)-\sum_{i=j+1}^{n-1}(\hat{a}_{n+j-i}s_i)-\sum_{i=0}^{j}(\epsilon_{a_{j-i}}s_i)+\sum_{i=j+1}^{n-1}(\epsilon_{a_{n+j-i}}s_i)+\hat{\Delta}m_j+(\hat{e}_j-\epsilon_{e_j})+\hat{q}\cdot c_j+\epsilon_{b_j}$
$=\left(\sum_{i=0}^{j}(\hat{a}_{j-i}s_i)-\sum_{i=j+1}^{n-1}(\hat{a}_{n+j-i}s_i)\right)+\hat{\Delta}m_j+\hat{e}_j+\left(\epsilon_{b_j}-\epsilon_{e_j}-\sum_{i=0}^{j}(\epsilon_{a_{j-i}}s_i)+\sum_{i=j+1}^{n-1}(\epsilon_{a_{n+j-i}}s_i)\right)+\hat{q}\cdot c_j$
$=\left(\sum_{i=0}^{j}(\hat{a}_{j-i}s_i)-\sum_{i=j+1}^{n-1}(\hat{a}_{n+j-i}s_i)\right)+\hat{\Delta}m_j+\hat{e}_j+\epsilon_{all}\in\mathbb{Z}_{\hat{q}}$
\# where $\epsilon_{all}=\epsilon_{b_j}-\epsilon_{e_j}-\sum_{i=0}^{j}(\epsilon_{a_{j-i}}s_i)+\sum_{i=j+1}^{n-1}(\epsilon_{a_{n+j-i}}s_i)\approx 0$

5. To summarize, for each $0\le j\le n-1$, each polynomial degree coefficient $\hat{b}_j$ is approximately as

follows:

$$\hat{b}_j = \left(\sum_{i=0}^{j} (\hat{a}_{j-i} s_i) - \sum_{i=j+1}^{n-1} (\hat{a}_{n+j-i} s_i) \right) + \hat{\Delta} m_j + \hat{e}_j + \epsilon_{all}$$

$$\approx \left(\sum_{i=0}^{j} (\hat{a}_{j-i} s_i) - \sum_{i=j+1}^{n-1} (\hat{a}_{n+j-i} s_i) \right) + \hat{\Delta} m_j + \hat{e}_j$$

Thus, $(\hat{a}_0, \hat{a}_1, \cdots, \hat{b}) = \mathsf{RLWE}_{S,\sigma}(\hat{\Delta} M)$, decrypting which will give us M.

□

C-4.5 GLWE Modulus Switching

GLWE modulus switching is an extension of RLWE modulus switching. The only difference is that while RLWE's A and S are a single polynomial each, GLWE's A and S are a list of k polynomials each. Thus, the same modulus switching technique as RLWE can be applied to GLWE for its k polynomials.

Recall that the GLWE cryptosystem **(??)** is comprised of the following components:

- **Initial Setup:** $\Delta = \frac{q}{t}, \quad \{S_i\}_{i=0}^{k-1} \xleftarrow{\$} \mathcal{R}_{\langle n,2 \rangle}^{k}$
- **Encryption Input:** $M \in \mathcal{R}_{\langle n,t \rangle}, \quad \{A_i\}_{i=0}^{k-1} \xleftarrow{\$} \mathcal{R}_{\langle n,q \rangle}^{k}, \quad E \xleftarrow{\chi_\sigma} \mathcal{R}_{\langle n,q \rangle}$
- **Encryption:** $\mathsf{GLWE}_{S,\sigma}(\Delta M) = (\{A_i\}_{i=0}^{k-1}, B) \in \mathcal{R}_{\langle n,q \rangle}^{k+1}$
  , where $B = \sum_{i=0}^{k-1} (A_i \cdot S_i) + \Delta M + E \in \mathcal{R}_{\langle n,q \rangle}$
- **Decryption:** $\mathsf{GLWE}_{S,\sigma}^{-1}(C) = B - \sum_{i=0}^{k-1} (A_i \cdot S_i) = \Delta M + E \in \mathcal{R}_{\langle n,q \rangle}$

GLWE modulus switching is done as follows:

⟨Summary ??⟩ GLWE Modulus Switching

Given a GLWE ciphertext $(\vec{A}, B)$ where $B = \vec{A} \cdot \vec{S} + \Delta M + E \bmod q$ and $M \in \mathcal{R}_{\langle n,q \rangle}$, the modulus switch of the ciphertext from q to $\hat{q}$ is equivalent to updating $(\vec{A}, B)$ to $(\hat{\vec{A}}, \hat{B})$ as follows:

$\hat{A}_i = \hat{a}_{i,0} + \hat{a}_{i,1} X + \hat{a}_{i,2} X^2 + \cdots + \hat{a}_{i,n-1} X^{n-1}$, where each $\hat{a}_{i,j} = \left\lceil a_{i,j} \frac{\hat{q}}{q} \right\rfloor \in \mathbb{Z}_{\hat{q}}$

$\hat{B} = \hat{b}_0 + \hat{b}_1 X + \hat{b}_2 X^2 + \cdots + \hat{b}_{n-1} X^{n-1}$, where each $\hat{b}_j = \left\lceil b_j \frac{\hat{q}}{q} \right\rfloor \in \mathbb{Z}_{\hat{q}}$

$\mathsf{GLWE}_{S,\sigma}(\hat{\Delta} M) = (\hat{A}, \hat{B}) \in \mathcal{R}_{\langle n,\hat{q} \rangle}^{k+1}$

The above update effectively changes E and Δ as follows:

$\hat{E} = \hat{e}_0 + \hat{e}_1 X + \hat{e}_2 X^2 + \cdots + \hat{e}_{n-1} X^{n-1}$, where each $\hat{e}_j = \left\lceil e_j \frac{\hat{q}}{q} \right\rfloor \in \mathbb{Z}_{\hat{q}}$

$\hat{\Delta} = \Delta \frac{\hat{q}}{q}$ # which should be an integer

Meanwhile, $\vec{S}$ and M stay the same as before.

The proof is similar to that of RLWE modulus switching. The modulus-switched GLWE ciphertext's culminating rounding drift error for each j-th polynomial coefficient in its congruence relationship (i.e., $B = \sum_{i=0}^{k-1} A_i \cdot S_i + \Delta M + E$) is as follows:

$$\epsilon_{j,all} = \epsilon_{b_j} - \epsilon_{e_j} - \sum_{l=0}^{k-1} \sum_{i=0}^{j} (\epsilon_{a_{l,j-i}} \cdot s_{l,i}) + \sum_{l=0}^{k-1} \sum_{i=j+1}^{n-1} (\epsilon_{a_{l,n+j-i}} \cdot s_{l,i})$$

derived from the proof step 4 of Summary **??**: $\epsilon_{all} = \epsilon_{b_j} - \epsilon_{e_j} - \sum_{i=0}^{j} (\epsilon_{a_{j-i}} s_i) + \sum_{i=j+1}^{n-1} (\epsilon_{a_{n+j-i}} s_i)$

Note that GLWE's modulus switching can have a bigger rounding drift error (about k times) than that of RLWE's modulus switching. However, in the long run, they can cancel out and converge to 0 as they are sampled from the σ distribution.

C-5 GLWE Key Switching

- Reference: TFHE Deep Dive - Part III - Key switching and leveled multiplications [**?**]

Key switching is a process to change a GLWE ciphertext's secret key from S to a new key S' without decrypting the ciphertext. This is equivalent to converting a ciphertext $\mathsf{GLWE}_{S,\sigma}(\Delta M)$ into a new ciphertext $\mathsf{GLWE}_{S',\sigma}(\Delta M)$.

Remember that $\mathsf{GLWE}_{S,\sigma}(\Delta M) = (A_0, A_1, \cdots, A_{k-1}, B) \in \mathcal{R}^{k+1}_{\langle n,q \rangle}$

, where $B = \sum_{i=0}^{k-1} (A_i \cdot S_i) + \Delta \cdot M + E$

, and the secret key S is a list of k polynomials: $S = (S_0, S_1, \cdots S_{k-1})$

Also, remember that GLev (**??**) is defined as follows:

$\mathsf{GLev}^{\beta,l}_{S,\sigma}(M) = \left\{ \mathrm{GLWE}_{S,\sigma}\left(\frac{q}{\beta^i} \cdot M\right) \right\}_{i=1}^{l} \in \mathcal{R}^{(k+1)\cdot l}_{\langle n,q \rangle}$

Now, let's denote each of the k key-switching keys as follows:

$KSK_i = \mathsf{GLev}^{\beta,l}_{S',\sigma}(S_i)$

$= \left(\mathsf{GLWE}_{S',\sigma}\left(\frac{q}{\beta^1} S_i\right), \mathsf{GLWE}_{S',\sigma}\left(\frac{q}{\beta^2} S_i\right), \cdots, \mathsf{GLWE}_{S',\sigma}\left(\frac{q}{\beta^l} S_i\right) \right) \in \mathcal{R}^{(k+1)\cdot l}_{\langle n,q \rangle}$

, which is a list of GLWE encryptions of the secret key S by S'. Then, the GLWE ciphertext's key can be switched from $S \to S'$ as follows:

⟨Summary ??⟩ GLWE Key Switching

$$\mathsf{GLWE}_{S',\sigma}(\Delta M) = (\overbrace{0, 0, \cdots 0}^{k}, B) - \sum_{i=0}^{k-1} A_i \cdot S_i$$
$$= (0, 0, \cdots 0, B) - \sum_{i=0}^{k-1} \langle \mathsf{Decomp}^{\beta,l}(A_i), KSK_i \rangle$$

Proof.

1. Given the principle of polynomial decomposition (**??**) and the polynomial $A_i \in \mathbb{Z}_q[x]/(x^n+1)$, its decomposition is as follows:

   $\mathsf{Decomp}^{\beta,l}(A_i) = (A_{\langle i,1 \rangle}, A_{\langle i,2 \rangle}, \cdots A_{\langle i,l \rangle})$, where

   $A_i = A_{\langle i,1 \rangle} \frac{q}{\beta^1} + A_{\langle i,2 \rangle} \frac{q}{\beta^2} + \cdots + A_{\langle i,l \rangle} \frac{q}{\beta^l}$

2. $\langle \mathsf{Decomp}^{\beta,l}(A_i), KSK_i \rangle$

   $= A_{\langle i,1 \rangle} \cdot \mathsf{GLWE}_{S',\sigma}\left(\frac{q}{\beta^1} S_i\right) + A_{\langle i,2 \rangle} \cdot \mathsf{GLWE}_{S',\sigma}\left(\frac{q}{\beta^2} S_i\right) + \cdots + A_{\langle i,l \rangle} \cdot \mathsf{GLWE}_{S',\sigma}\left(\frac{q}{\beta^l} S_i\right)$ # where each GLWE ciphertext is an encryption of $S_i \frac{q}{\beta}, S_i \frac{q}{\beta^2}, \cdots, S_i \frac{q}{\beta^l}$ as plaintext with the plaintext scaling factor 1

$= \mathsf{GLWE}_{S',\sigma}\left(\left(A_{\langle i,1\rangle}\frac{q}{\beta^1} + A_{\langle i,2\rangle}\frac{q}{\beta^2} + \cdots A_{\langle i,l\rangle}\frac{q}{\beta^l}\right)\cdot S_i\right)$
$= \mathsf{GLWE}_{S',\sigma}\left((A_i)\cdot S_i\right) = \mathsf{GLWE}_{S',\sigma}(A_iS_i)$

3. $\sum_{i=0}^{k-1}\langle \mathsf{Decomp}^{\beta,l}(A_i), KSK_i\rangle = \sum_{i=0}^{k-1}(\mathsf{GLWE}_{S',\sigma}(A_iS_i))$
$= \mathsf{GLWE}_{S',\sigma}\left(\sum_{i=0}^{k-1} A_iS_i\right)$

4. B is equivalent to a trivial GLWE encryption with S', where its every $A_1, A_2, \cdots$ is 0. That is, $\mathsf{GLWE}_{S',\sigma}(B) = (\overbrace{0, 0, \cdots}^{k}, B)$. Note that $\mathsf{GLWE}_{S',\sigma}(B)$ can be created without the knowledge of S', because its all A_iS_i terms are 0.

5. $\mathsf{GLWE}_{S',\sigma}(B) - \mathsf{GLWE}_{S',\sigma}\left(\sum_{i=0}^{k-1}(A_iS_i)\right) = \mathsf{GLWE}_{S',\sigma}(B - \sum_{i=0}^{k-1}(A_iS_i)) = \mathsf{GLWE}_{S',\sigma}(\Delta M + E)$

6. To expand the above relation,
$\mathsf{GLWE}_{S',\sigma}(B) - \mathsf{GLWE}_{S',\sigma}\left(\sum_{i=0}^{k-1}(A_iS_i)\right)$
$= (\overbrace{0, 0, \cdots}^{k}, B) - (\overbrace{A'_0, A'_1, \cdots, A'_k}^{k}, B') \quad \#\ \text{where } B' = \sum_{i=0}^{k-1} A'_iS'_i + \sum_{i=0}^{k-1} A_iS_i + E'$
$(\overbrace{-A'_0, -A'_1, \cdots, -A'_k}^{k},\ B - B')$

   The decryption of the above ciphertext gives us:
$\mathsf{GLWE}^{-1}_{S',\sigma}\left(\ (\overbrace{-A'_0, -A'_1, \cdots, -A'_k}^{k},\ B - B')\ \right)$
$= B - B' - \sum_{i=0}^{k-1} -A'_iS'_i$
$= B - \sum_{i=0}^{k-1} A'_iS'_i - \sum_{i=0}^{k-1} A_iS_i - E' + \sum_{i=0}^{k-1} A'_iS'_i$
$= B - \sum_{i=0}^{k-1} A_iS_i - E'$
$= \Delta M + E - E' \approx \Delta M + E$
   Strictly speaking, $B - \sum_{i=0}^{k-1} A_iS_i = \Delta M + E + Kq$ where K is a polynomial to represent the wrap-around coefficient values as multiples of q. However, since all the above computations are done in modulo q, the Kq term gets eliminated.

7. Thus, $(0, 0, \cdots, B) - \sum_{i=0}^{k-1}\langle \mathsf{Decomp}^{\beta,l}(A_i), KSK_i\rangle \approx \mathsf{GLWE}_{S',\sigma}(\Delta M)$
   $\mathsf{GLWE}_{S',\sigma}(\Delta M)$ is an encryption of plaintext ΔM with the plaintext scaling factor 1. However, we can theoretically see this as an encryption of plaintext M with the plaintext scaling factor Δ. This way, we can recover the ciphertext's original scaling factor Δ without any additional

computation.

□

Part IV
Fully Homomorphic Encryption Schemes

This chapter explains the four most well-known FHE schemes: TFHE, CKKS, BGV, and BFV, as well as their RNS-variant versions.

D-1 TFHE Scheme

The TFHE scheme is designed for homomorphic addition and multiplication on integers (especially bit-wise computation, like logic circuits). Unlike BFV, GBV, or CKKS, TFHE is characterized by fast noise bootstrapping; therefore, it is efficient for processing deep multiplication depths. TFHE's noise bootstrapping technique can be further applied to functional encryption.

In TFHE, each plaintext is encrypted as an LWE ciphertext. Therefore, TFHE's ciphertext-to-ciphertext addition, ciphertext-to-plaintext addition, and ciphertext-to-plaintext multiplication are implemented based on GLWE's homomorphic addition and multiplication described in **??**, with $n = 1$ to make GLWE an LWE.

This section will explain TFHE's novel components: key switching, ciphertext-to-ciphertext multiplication, coefficient extraction, and noise bootstrapping.

Required Background

- ??: ??
- ??: ??
- ??: ??
- ??: ??
- ??: ??
- ??: ??
- ??: ??
- ??: ??
- ??: ??
- ??: ??
- ??: ??
- ??: ??
- ??: ??
- ??: ??
- ??: ??
- ??: ??
- ??: ??
- ??: ??

D-1.1 Encryption and Decryption

TFHE encrypts and decrypts ciphertexts based on the LWE cryptosystem (**??**), which is equivalent to the GLWE cryptosystem (**??**) with $n = 1$.

⟨Summary ??⟩ TFHE Encryption and Decryption

Initial Setup: $\Delta = \frac{q}{t}$, $\vec{s} \xleftarrow{\$} \mathbb{Z}_2^k$ # where t divides q, and each element of $\vec{s}$ is a 0-degree polynomial

---

Encryption Input: $m \in \mathbb{Z}_t$, $\vec{a} \xleftarrow{\$} \mathbb{Z}_q^k$, $e \xleftarrow{\chi_\sigma} \mathbb{Z}_q$ # each element of $\vec{a}$ is a 0-degree polynomial

1. Scale up $m \longrightarrow \Delta \cdot m \ \in \mathbb{Z}_q$
2. Compute $b = \vec{a} \cdot \vec{s} + \Delta m + e \mod q$
3. $\mathsf{LWE}_{\vec{s},\sigma}(\Delta m) = (\vec{a}, b) \ \in \mathbb{Z}_q^{k+1}$

Decryption Input: $\mathsf{ct} = (\vec{a}, b) \ \in \mathbb{Z}_q^{k+1}$

1. $\mathsf{LWE}_{\vec{s},\sigma}^{-1}(\mathsf{ct}) = b - \vec{a} \cdot \vec{s} = \Delta m + e \pmod q$
2. Scale down $\left\lceil \frac{\Delta m + e}{\Delta} \right\rfloor = m \ \in \mathbb{Z}_t$ # i.e., modulus switch from $q \to t$

Condition for Correct Decryption:

- The noise e grown over homomorphic operations should be: $e < \frac{\Delta}{2}$.

D-1.2 Homomorphic Ciphertext-to-Ciphertext Addition

TFHE's ciphertext-to-ciphertext addition uses LWE's ciphertext-to-ciphertext addition scheme, which is equivalent to GLWE's ciphertext-to-ciphertext addition scheme (**??**) with $n = 1$.

⟨Summary ??⟩ TFHE Ciphertext-to-Ciphertext Addition

$$\begin{aligned}
&\mathsf{LWE}_{\vec{s},\sigma}(\Delta m^{\langle 1 \rangle}) + \mathsf{LWE}_{\vec{s},\sigma}(\Delta m^{\langle 2 \rangle}) \\
&= (\vec{a}^{\langle 1 \rangle},\ b^{\langle 1 \rangle}) + (\vec{a}^{\langle 2 \rangle},\ b^{\langle 2 \rangle}) \\
&= (\vec{a}^{\langle 1 \rangle} + \vec{a}^{\langle 2 \rangle},\ b^{\langle 1 \rangle} + b^{\langle 2 \rangle}) \\
&= \mathsf{LWE}_{\vec{s},\sigma}(\Delta(m^{\langle 1 \rangle} + m^{\langle 2 \rangle}))
\end{aligned}$$

D-1.3 Homomorphic Ciphertext-to-Plaintext Addition

TFHE's ciphertext-to-plaintext addition (where λ is a constant to add) uses LWE's ciphertext-to-plaintext addition scheme, which is equivalent to GLWE's ciphertext-to-plaintext addition scheme (**??**) with $n = 1$.

⟨Summary ??⟩ TFHE Ciphertext-to-Plaintext Addition

$$\begin{aligned}&\mathsf{LWE}_{\vec{s},\sigma}(\Delta m) + \Delta\lambda\\ &= (\vec{a},\ B) + \Delta\lambda\\ &= (\vec{a},\ B + \Delta\lambda)\\ &= \mathsf{LWE}_{\vec{s},\sigma}(\Delta(m+\lambda))\end{aligned}$$

D-1.4 Homomorphic Ciphertext-to-Plaintext Multiplication

TFHE's ciphertext-to-plaintext multiplication uses LWE's ciphertext-to-plaintext multiplication scheme, which is equivalent to GLWE's ciphertext-to-plaintext multiplication scheme (**??**) with $n = 1$.

⟨Summary ??⟩ TFHE Ciphertext-to-Plaintext Multiplication

$$\begin{aligned}&\mathsf{LWE}_{\vec{s},\sigma}(\Delta m) \cdot \lambda\\ &= (\vec{a},\ b) \cdot \lambda\\ &= (\lambda \cdot \vec{a},\ \lambda \cdot b)\\ &= \mathsf{LWE}_{\vec{s},\sigma}(\Delta(m \cdot \lambda))\end{aligned}$$

D-1.5 Homomorphic Key Switching

- Reference: TFHE Deep Dive - Part III - Key switching and leveled multiplications [**?**]

TFHE's key switching scheme changes an LWE ciphertext's secret key from $\vec{s}$ to $\vec{s\prime}$. This scheme is essentially LWE's key switching scheme. Specifically, this is equivalent to the alternative GLWE version's (**??**) key switching scheme (**??**) with $n = 1$ as follows:

⟨Summary ??⟩ TFHE Key Switching

$$\mathsf{LWE}_{\vec{s\prime},\sigma}(\Delta m) = (0, b) + \langle \mathsf{Decomp}^{\beta,l}(\vec{a}),\ \mathsf{Lev}^{\beta,l}_{\vec{s\prime},\sigma}(\vec{s}) \rangle$$

D-1.6 Homomorphic Ciphertext-to-Ciphertext Multiplication

- Reference: TFHE Deep Dive - Part III - Key switching and leveled multiplications [**?**]

TFHE supports multiplication of two ciphertexts in the form: $\mathsf{LWE}_{\vec{s},\sigma}(\Delta m_1) \cdot \mathsf{GSW}^{\beta,l}_{\vec{s},\sigma}(m_2)$.

The 1st term $\mathsf{LWE}_{\vec{s},\sigma}(\Delta m_1)$ comes from one of the following:

- A fresh LWE encryption (**??**) of plaintext m_1.
- A homomorphically added result of two LWE ciphertexts (**??**).
- A homomorphically multiplied result of a LWE ciphertext with a plaintext (**??**).

The 2nd term $\mathsf{GSW}^{\beta,l}_{\vec{s},\sigma}(m_2)$ comes from one of the following:

- A fresh GSW encryption (**??**) of plaintext m_2.

- Converted from $\mathsf{LWE}_{\vec{s},\sigma}(\Delta m_2)$ into $\mathsf{GSW}^{\beta,l}_{\vec{s},\sigma}(m_2)$ by *circuit bootstrapping* (this will be covered in the future).

Remember the following:

$\mathsf{LWE}_{\vec{s},\sigma}(\Delta m_1) = (\vec{a}, b) \in \mathbb{Z}_q^{k+1}$, where $b = \vec{a} \cdot \vec{s} + \Delta m_1 + e$

$\mathsf{GSW}^{\beta,l}_{\vec{s},\sigma}(m_2) = \left\{ \{\mathsf{Lev}^{\beta,l}_{\vec{s},\sigma}(-s_i \cdot m_2)\}_{i=0}^{k-1}, \mathsf{Lev}^{\beta,l}_{\vec{s},\sigma}(m_2) \right\} \in \mathbb{Z}_q^{(k+1)\cdot l \cdot (k+1)}$ # from **??**

Let's use the following notations:

$\mathsf{GSW}^{\beta,l}_{\vec{s},\sigma}(m_2) = \bar{\mathsf{ct}} = (\bar{\mathsf{ct}}_0, .\bar{\mathsf{ct}}_1, \ \cdots \ \bar{\mathsf{ct}}_k)$

$\bar{\mathsf{ct}}_i = \mathsf{Lev}^{\beta,l}_{\vec{s},\sigma}(-s_i \cdot m_2)$ for $0 \leq i \leq (k-1)$

$\bar{\mathsf{ct}}_k = \mathsf{Lev}^{\beta,l}_{\vec{s},\sigma}(m_2)$

$\mathsf{ct} = \mathsf{LWE}_{\vec{s},\sigma}(\Delta m_1) = (\vec{a}, b) = (a_0, a_1, \ \cdots \ , a_{k-1}, b) = (\mathsf{ct}_0, \mathsf{ct}_1, \cdots, \mathsf{ct}_k)$

Let's define the following TFHE ciphertext multiplication operation:

$$\mathsf{ct} \cdot \bar{\mathsf{ct}} = \sum_{i=0}^{k} \langle \mathsf{Decomp}^{\beta,l}(\mathsf{ct}_i), \bar{\mathsf{ct}}_i \rangle$$

Then, the following is true:

⟨Summary ??⟩ TFHE Ciphertext-to-Ciphertext Multiplication

$\mathsf{ct} = \mathsf{LWE}_{\vec{s},\sigma}(\Delta m_1) = (a_0, a_1, \cdots, a_{k-1}, b)$

$\bar{\mathsf{ct}} = \mathsf{GSW}^{\beta,l}_{\vec{s},\sigma}(m_2) = (\mathsf{Lev}^{\beta,l}_{\vec{s},\sigma}(-s_0 \cdot m_2), \mathsf{Lev}^{\beta,l}_{\vec{s},\sigma}(-s_1 \cdot m_2), \cdots, \mathsf{Lev}^{\beta,l}_{\vec{s},\sigma}(-s_{k-1} \cdot m_2), \mathsf{Lev}^{\beta,l}_{\vec{s},\sigma}(m_2))$

$$\mathsf{LWE}_{\vec{s},\sigma}(\Delta m_1) \cdot \mathsf{GSW}^{\beta,l}_{\vec{s},\sigma}(m_2) = \sum_{i=0}^{k} \langle \mathsf{Decomp}^{\beta,l}(\mathsf{ct}_i), \bar{\mathsf{ct}}_i \rangle = \mathsf{LWE}_{\vec{s},\sigma}(\Delta m_1 m_2)$$

This means that multiplying two TFHE ciphertexts (one is in LWE and another in GSW) and decrypting the resulting LWE ciphertext gives the same result as multiplying their two original plaintexts.

Proof.

1. $\sum_{i=0}^{k} \langle \mathsf{Decomp}^{\beta,l}(\mathsf{ct}_i), \bar{\mathsf{ct}}_i \rangle$
   $= \langle \mathsf{Decomp}^{\beta,l}(a_0), \bar{\mathsf{ct}}_0 \rangle + \langle \mathsf{Decomp}^{\beta,l}(a_1), \bar{\mathsf{ct}}_1 \rangle + \ \cdots \ + \langle \mathsf{Decomp}^{\beta,l}(a_{k-1}), \bar{\mathsf{ct}}_{k-1} \rangle + \langle \mathsf{Decomp}^{\beta,l}(b), \bar{\mathsf{ct}}_k \rangle$
   # expanding the dot product of two vectors
2. For $i = k$:
   $\mathsf{Decomp}^{\beta,l}(b) = (b_1, b_2, \ \cdots \ , b_l)$, where $b = b_1 \frac{q}{\beta^1} + b_2 \frac{q}{\beta^2} + \ \cdots \ + b_l \frac{q}{\beta^l}$ # from **??**

   $$\bar{\mathsf{ct}}_k = \mathsf{Lev}^{\beta,l}_{\vec{s},\sigma}(m_2) = \left(\mathsf{LWE}_{\vec{s},\sigma}\left(m_2 \frac{q}{\beta^1}\right), \mathsf{LWE}_{\vec{s},\sigma}\left(m_2 \frac{q}{\beta^2}\right), \ \cdots \ , \mathsf{LWE}_{\vec{s},\sigma}\left(m_2 \frac{q}{\beta^l}\right) \right)$$

   Therefore:
   $\langle \mathsf{Decomp}^{\beta,l}(b), \bar{\mathsf{ct}}_k \rangle$
   $$= b_1 \cdot \mathsf{LWE}_{\vec{s},\sigma}\left(m_2 \frac{q}{\beta^1}\right) + b_2 \cdot \mathsf{LWE}_{\vec{s},\sigma}\left(m_2 \frac{q}{\beta^2}\right) + \ \cdots \ + b_l \cdot \mathsf{LWE}_{\vec{s},\sigma}\left(m_2 \frac{q}{\beta^l}\right)$$

$= \mathsf{LWE}_{\vec{s},\sigma}\left(b_1 m_2 \frac{q}{\beta^1}\right) + \mathsf{LWE}_{\vec{s},\sigma}\left(b_2 m_2 \frac{q}{\beta^2}\right) + \cdots + \mathsf{LWE}_{\vec{s},\sigma}\left(b_l m_2 \frac{q}{\beta^l}\right)$ # from **??**

$= \mathsf{LWE}_{\vec{s},\sigma}\left(b_1 m_2 \frac{q}{\beta^1} + b_2 m_2 \frac{q}{\beta^2} + \cdots + b_l m_2 \frac{q}{\beta^l}\right)$ # from **??**

$= \mathsf{LWE}_{\vec{s},\sigma}\left(m_2 \cdot \left(b_1 \frac{q}{\beta^1} + b_2 \frac{q}{\beta^2} + \cdots + b_l \frac{q}{\beta^l}\right)\right)$

$= \mathsf{LWE}_{\vec{s},\sigma}(m_2 b)$ # from **??**

3. For $0 \leq i \leq (k-1)$:

   $\mathsf{Decomp}^{\beta,l}(a_i) = (a_{\langle i,1\rangle}, a_{\langle i,2\rangle}, \cdots, a_{\langle i,l\rangle})$, where $a_i = a_{\langle i,1\rangle}\frac{q}{\beta^1} + a_{\langle i,2\rangle}\frac{q}{\beta^2} + \cdots + a_{\langle i,l\rangle}\frac{q}{\beta^l}$

   $\bar{\mathsf{ct}}_i = \mathsf{Lev}^{\beta,l}_{\vec{s},\sigma}(-s_i m_2) = \left(\mathsf{LWE}_{\vec{s},\sigma}\left(-s_i m_2 \frac{q}{\beta^1}\right), \mathsf{LWE}_{\vec{s},\sigma}\left(-s_i m_2 \frac{q}{\beta^2}\right), \cdots, \mathsf{LWE}_{\vec{s},\sigma}\left(-s_i m_2 \frac{q}{\beta^l}\right)\right)$

   Therefore:

   $\langle \mathsf{Decomp}^{\beta,l}(a_0), \bar{\mathsf{ct}}_0\rangle + \langle \mathsf{Decomp}^{\beta,l}(a_1), \bar{\mathsf{ct}}_1\rangle + \cdots + \langle \mathsf{Decomp}^{\beta,l}(a_{k-1}), \bar{\mathsf{ct}}_{k-1}\rangle$

   $= \sum_{i=0}^{k-1} \langle \mathsf{Decomp}^{\beta,l}(a_i), \bar{\mathsf{ct}}_i\rangle$

   $= \sum_{i=0}^{k-1} \left(a_{\langle i,1\rangle} \cdot \mathsf{LWE}_{\vec{s},\sigma}\left(-s_i m_2 \frac{q}{\beta^1}\right) + a_{\langle i,2\rangle} \cdot \mathsf{LWE}_{\vec{s},\sigma}\left(-s_i m_2 \frac{q}{\beta^2}\right) + \cdots + a_{\langle i,l\rangle} \cdot \mathsf{LWE}_{\vec{s},\sigma}\left(-s_i m_2 \frac{q}{\beta^l}\right)\right)$

   $= \sum_{i=0}^{k-1} \left(\mathsf{LWE}_{\vec{s},\sigma}\left(-a_{\langle i,1\rangle} s_i m_2 \frac{q}{\beta^1}\right) + \mathsf{LWE}_{\vec{s},\sigma}\left(-a_{\langle i,2\rangle} s_i m_2 \frac{q}{\beta^2}\right) + \cdots + \mathsf{LWE}_{\vec{s},\sigma}\left(-a_{\langle i,l\rangle} s_i m_2 \frac{q}{\beta^l}\right)\right)$

   $= \sum_{i=0}^{k-1} \mathsf{LWE}_{\vec{s},\sigma}\left(-a_{\langle i,1\rangle} s_i m_2 \frac{q}{\beta^1} + -a_{\langle i,2\rangle} s_i m_2 \frac{q}{\beta^2} + \cdots + -a_{\langle i,l\rangle} s_i m_2 \frac{q}{\beta^l}\right)$

   $= \sum_{i=0}^{k-1} \mathsf{LWE}_{\vec{s},\sigma}\left(-s_i m_2 \cdot \left(a_{\langle i,1\rangle} \frac{q}{\beta^1} + a_{\langle i,2\rangle} \frac{q}{\beta^2} + \cdots + a_{\langle i,l\rangle} \frac{q}{\beta^l}\right)\right)$

   $= \sum_{i=0}^{k-1} \mathsf{LWE}_{\vec{s},\sigma}(-s_i m_2 a_i)$

4. According to step 2 and 3,

   $\sum_{i=0}^{k} \langle \mathsf{Decomp}^{\beta,l}(\mathsf{ct}_i), \bar{\mathsf{ct}}_i\rangle$

   $= \sum_{i=0}^{k-1} \mathsf{LWE}_{\vec{s},\sigma}(-s_i m_2 a_i) + \mathsf{LWE}_{\vec{s},\sigma}(m_2 b)$

   $= \mathsf{LWE}_{\vec{s},\sigma}\left(\sum_{i=0}^{k-1}(-s_i m_2 a_i) + m_2 b\right)$ # addition of two GLWE ciphertexts

   $= \mathsf{LWE}_{\vec{s},\sigma}\left(m_2 b - \sum_{i=0}^{k-1} m_2 a_i s_i\right)$

   $= \mathsf{LWE}_{\vec{s},\sigma}\left(m_2 (b - \sum_{i=0}^{k-1} a_i s_i)\right)$

   $= \mathsf{LWE}_{\vec{s},\sigma}(m_2(\Delta m_1 + e))$

   $= \mathsf{LWE}_{\vec{s},\sigma}(\Delta m_1 m_2 + m_2 e)$

   $\approx \mathsf{LWE}_{\vec{s},\sigma}(\Delta m_1 m_2)$ # given e is small and thus $m_2 e$ is also small

□

D-1.6.1 Discussion on the Noise Growth

Note that after ciphertext-to-ciphertext multiplication, the noise grows to:

$$\mathsf{LWE}_{\vec{s},\sigma}(\Delta m_1) \cdot \mathsf{GSW}^{\beta,l}_{S,\sigma}(m_2) = \mathsf{LWE}_{\vec{s},\sigma}(\Delta m_1 m_2 + m_2 e) \ (\approx \mathsf{LWE}_{\vec{s},\sigma}(\Delta m_1 m_2))$$

To reduce the noise, noise bootstrapping is needed (will be discussed in **??**).

D-1.6.2 Generalization to GLWE-to-GGSW Multiplication

We can further generalize TFHE's LWE-to-GSW multiplication to GLWE-to-GGSW multiplication between the following two ciphertexts: $\mathsf{GLWE}_{\vec{S},\sigma}(\Delta M_1) \cdot \mathsf{GGSW}^{\beta,l}_{\vec{S},\sigma}(M_2)$, where M_1, M_2, and S are $(n-1)$-degree polynomials.

The 1st term $\mathsf{GLWE}_{\vec{S},\sigma}(\Delta M_1)$ comes from one of the following:
- A fresh GLWE encryption (**??**) of plaintext M_1.
- A homomorphically added result of two GLWE ciphertexts (**??**).
- A homomorphically multiplied result of a GLWE ciphertext with a plaintext (**??**).

The 2nd term $\mathsf{GGSW}^{\beta,l}_{\vec{S},\sigma}(M_2)$ comes from one of the following:
- A fresh GGSW encryption (**??**) of plaintext M_2.
- Converted from $\mathsf{GLWE}_{\vec{S},\sigma}(\Delta M_2)$ into $\mathsf{GGSW}^{\beta,l}_{\vec{S},\sigma}(M_2)$ by *circuit bootstrapping* (this will be covered in the future).

Remember the following:

$$\mathsf{GLWE}_{\vec{S},\sigma}(\Delta M_1) = (A_0, A_0, \ \cdots \ , A_{k-1}, B) \in \mathcal{R}^{k+1}_{n,q}, \text{ where } B = \sum_{i=0}^{k-1}(A_i \cdot S_i) + \Delta M_1 + E$$

from **??**

$$\mathsf{GGSW}^{\beta,l}_{\vec{S},\sigma}(M_2) = \left\{ \{\mathsf{GLev}^{\beta,l}_{\vec{S},\sigma}(-S_i \cdot M_2)\}_{i=0}^{k-1}, \mathsf{GLev}^{\beta,l}_{\vec{S},\sigma}(M_2) \right\} \in \mathcal{R}^{(k+1)\cdot l\cdot(k+1)}_{\langle n,q\rangle} \ \text{\# from **??**}$$

Let's use the following notations:

$$\mathsf{GGSW}^{\beta,l}_{\vec{S},\sigma}(M_2) = \bar{C} = (\bar{C_0}, .\bar{C_1}, \ \cdots \ \bar{C_k})$$
$$\bar{C_i} = \mathsf{GLev}^{\beta,l}_{\vec{S},\sigma}(-S_i \cdot M_2) \text{ for } 0 \leq i \leq (k-1)$$
$$\bar{C_k} = \mathsf{GLev}^{\beta,l}_{\vec{S},\sigma}(M_2)$$
$$\mathsf{ct} = \mathsf{GLWE}_{\vec{S},\sigma}(\Delta M_1) = (C_0, C_1, \ \cdots \ , C_k) = (A_0, A_1, \ \cdots \ , A_{k-1}, B)$$

Let's define the following TFHE ciphertext multiplication operation:

$$\mathsf{ct} \cdot \bar{C} = \sum_{i=0}^{k} \langle \mathsf{Decomp}^{\beta,l}(C_i), \bar{C_i} \rangle$$

Then, the following is true:

⟨Summary ??⟩ Generalization to GLWE-to-GGSW Multiplication

$$\mathsf{ct} = \mathsf{GLWE}_{\vec{S},\sigma}(\Delta M_1) = (A_0, A_1, \cdots, A_{k-1}, B)$$
$$\bar{C} = \mathsf{GGSW}^{\beta,l}_{\vec{S},\sigma}(M_2)$$
$$= (\mathsf{GLev}^{\beta,l}_{\vec{S},\sigma}(-S_0 \cdot M_2), \mathsf{GLev}^{\beta,l}_{\vec{S},\sigma}(-S_1 \cdot M_2), \cdots, \mathsf{GLev}^{\beta,l}_{\vec{S},\sigma}(-S_{k-1} \cdot M_2), \mathsf{GLev}^{\beta,l}_{\vec{S},\sigma}(M_2))$$

$$\mathsf{GLWE}_{\vec{S},\sigma}(\Delta M_1) \cdot \mathsf{GGSW}^{\beta,l}_{\vec{S},\sigma}(M_2) = \sum_{i=0}^{k} \langle \mathsf{Decomp}^{\beta,l}(C_i), \bar{C}_i \rangle = \mathsf{GLWE}_{\vec{S},\sigma}(\Delta M_1 M_2)$$

This means that multiplying two TFHE ciphertexts (one is in GLWE and another in GGSW) and decrypting the resulting GLWE ciphertext gives the same result as multiplying their two original plaintexts.

Proof.

1. $\sum_{i=0}^{k} \langle \mathsf{Decomp}^{\beta,l}(C_i), \bar{C}_i \rangle$
   $= \langle \mathsf{Decomp}^{\beta,l}(A_0), \bar{C}_0 \rangle + \langle \mathsf{Decomp}^{\beta,l}(A_1), \bar{C}_1 \rangle + \cdots + \langle \mathsf{Decomp}^{\beta,l}(A_{k-1}), \bar{C}_{k-1} \rangle + \langle \mathsf{Decomp}^{\beta,l}(B), \bar{C}_k \rangle$
   # expanding the dot product of two vectors
2. For $i = k$:
   $\mathsf{Decomp}^{\beta,l}(B) = (B_1, B_2, \cdots, B_l)$, where $B = B_1 \frac{q}{\beta^1} + B_2 \frac{q}{\beta^2} + \cdots + B_l \frac{q}{\beta^l}$ # from **??**
   $\bar{C}_k = \mathsf{GLev}^{\beta,l}_{\vec{S},\sigma}(M_2) = \left(\mathsf{GLWE}_{\vec{S},\sigma}\left(M_2 \frac{q}{\beta^1}\right), \mathsf{GLWE}_{\vec{S},\sigma}\left(M_2 \frac{q}{\beta^2}\right), \cdots, \mathsf{GLWE}_{\vec{S},\sigma}\left(M_2 \frac{q}{\beta^l}\right) \right)$

   Therefore:
   $\langle \mathsf{Decomp}^{\beta,l}(B), \bar{C}_k \rangle$
   $= B_1 \cdot \mathsf{GLWE}_{\vec{S},\sigma}\left(M_2 \frac{q}{\beta^1}\right) + B_2 \cdot \mathsf{GLWE}_{\vec{S},\sigma}\left(M_2 \frac{q}{\beta^2}\right) + \cdots + B_l \cdot \mathsf{GLWE}_{\vec{S},\sigma}\left(M_2 \frac{q}{\beta^l}\right)$
   $= \mathsf{GLWE}_{\vec{S},\sigma}\left(B_1 M_2 \frac{q}{\beta^1}\right) + \mathsf{GLWE}_{\vec{S},\sigma}\left(B_2 M_2 \frac{q}{\beta^2}\right) + \cdots + \mathsf{GLWE}_{\vec{S},\sigma}\left(B_l M_2 \frac{q}{\beta^l}\right)$ # from **??**
   $= \mathsf{GLWE}_{\vec{S},\sigma}\left(B_1 M_2 \frac{q}{\beta^1} + B_2 M_2 \frac{q}{\beta^2} + \cdots + B_l M_2 \frac{q}{\beta^l}\right)$ # from **??**
   $= \mathsf{GLWE}_{\vec{S},\sigma}\left(M_2 \cdot \left(B_1 \frac{q}{\beta^1} + B_2 \frac{q}{\beta^2} + \cdots + B_l \frac{q}{\beta^l}\right)\right)$
   $= \mathsf{GLWE}_{\vec{S},\sigma}(M_2 B)$ # from **??**
3. For $0 \leq i \leq (k-1)$:
   $\mathsf{Decomp}^{\beta,l}(A_i) = (A_{\langle i,0 \rangle}, A_{\langle i,1 \rangle}, \cdots, A_{\langle i,l \rangle})$, where $A_i = A_{\langle i,0 \rangle} \frac{q}{\beta^1} + A_{\langle i,1 \rangle} \frac{q}{\beta^2} + \cdots + A_{\langle i,l \rangle} \frac{q}{\beta^l}$
   $\bar{C}_i = \mathsf{GLev}^{\beta,l}_{\vec{S},\sigma}(-S_i M_2) = \left(\mathsf{GLWE}_{\vec{S},\sigma}\left(-S_i M_2 \frac{q}{\beta^1}\right), \mathsf{GLWE}_{\vec{S},\sigma}\left(-S_i M_2 \frac{q}{\beta^2}\right), \cdots, \mathsf{GLWE}_{\vec{S},\sigma}\left(-S_i M_2 \frac{q}{\beta^l}\right) \right)$

   Therefore:
   $\langle \mathsf{Decomp}^{\beta,l}(A_0), \bar{C}_0 \rangle + \langle \mathsf{Decomp}^{\beta,l}(A_1), \bar{C}_1 \rangle + \cdots + \langle \mathsf{Decomp}^{\beta,l}(A_{k-1}), \bar{C}_{k-1} \rangle$
   $= \sum_{i=0}^{k-1} \langle \mathsf{Decomp}^{\beta,l}(A_i), \bar{C}_i \rangle$
   $= \sum_{i=0}^{k-1} \left(A_{\langle i,1 \rangle} \cdot \mathsf{GLWE}_{\vec{S},\sigma}\left(-S_i M_2 \frac{q}{\beta^1}\right) + A_{\langle i,2 \rangle} \cdot \mathsf{GLWE}_{\vec{S},\sigma}\left(-S_i M_2 \frac{q}{\beta^2}\right) + \cdots + A_{\langle i,l \rangle} \cdot \mathsf{GLWE}_{\vec{S},\sigma}\left(-S_i M_2 \frac{q}{\beta^l}\right) \right)$
   $= \sum_{i=0}^{k-1} \left(\mathsf{GLWE}_{\vec{S},\sigma}\left(-A_{\langle i,1 \rangle} S_i M_2 \frac{q}{\beta^1}\right) + \mathsf{GLWE}_{\vec{S},\sigma}\left(-A_{\langle i,2 \rangle} S_i M_2 \frac{q}{\beta^2}\right) + \cdots + \mathsf{GLWE}_{\vec{S},\sigma}\left(-A_{\langle i,l \rangle} S_i M_2 \frac{q}{\beta^l}\right) \right)$
   $= \sum_{i=0}^{k-1} \mathsf{GLWE}_{\vec{S},\sigma}\left(-A_{\langle i,1 \rangle} S_i M_2 \frac{q}{\beta^1} + -A_{\langle i,2 \rangle} S_i M_2 \frac{q}{\beta^2} + \cdots + -A_{\langle i,l \rangle} S_i M_2 \frac{q}{\beta^l}\right)$
   $= \sum_{i=0}^{k-1} \mathsf{GLWE}_{\vec{S},\sigma}\left(-S_i M_2 \cdot \left(A_{\langle i,1 \rangle} \frac{q}{\beta^1} + A_{\langle i,2 \rangle} \frac{q}{\beta^2} + \cdots + A_{\langle i,l \rangle} \frac{q}{\beta^l}\right)\right)$

$= \sum_{i=0}^{k-1} \mathsf{GLWE}_{\vec{S},\sigma}(-S_i M_2 A_i)$

4. According to step 2 and 3,

$\sum_{i=0}^{k=k} \langle \mathsf{Decomp}^{\beta,l}(C_i), \bar{C}_i \rangle$

$= \sum_{i=0}^{k-1} \mathsf{GLWE}_{\vec{S},\sigma}(-S_i M_2 A_i) + \mathsf{GLWE}_{\vec{S},\sigma}(M_2 B)$

$= \mathsf{GLWE}_{\vec{S},\sigma}\Big(\sum_{i=0}^{k-1} (-S_i M_2 A_i) + M_2 B \Big)$ # addition of two GLWE ciphertexts

$= \mathsf{GLWE}_{\vec{S},\sigma}\Big(B M_2 - \sum_{i=0}^{k-1} M_2 A_i S_i \Big)$

$= \mathsf{GLWE}_{\vec{S},\sigma}\Big(M_2 (B - \sum_{i=0}^{k-1} A_i S_i) \Big)$

$= \mathsf{GLWE}_{\vec{S},\sigma}(M_2(\Delta M_1 + E))$

$= \mathsf{GLWE}_{\vec{S},\sigma}(\Delta M_1 M_2 + M_2 E)$

$\approx \mathsf{GLWE}_{\vec{S},\sigma}(\Delta M_1 M_2)$ # given E is small and thus $M_2 E$ is also small

□

D-1.7 Coefficient Extraction

- Reference: TFHE Deep Dive - Part IV - Programmable Bootstrapping [**?**]

In TFHE, coefficient extraction is a process of extracting a coefficient of a polynomial that is encrypted as GLWE ciphertext. The extracted coefficient is in the form of LWE ciphertext (**??**).

Note that in the GLWE cryptosystem, plaintext M is encoded as a polynomial, where each coefficient encodes the plaintext value $m_0, m_1, \cdots, m_{n-1}$.

Suppose we have a GLWE ciphertext setup as the following:

$M = \sum_{j=0}^{n-1} m_j X^j \in \mathcal{R}_{\langle n,q \rangle}$

$S = \left(S_0 = \sum_{j=0}^{n-1} s_{0,j} X^j, S_1 = \sum_{j=0}^{n-1} s_{1,j} X^j, \ \cdots \ , S_{k-1} = \sum_{j=0}^{n-1} s_{k-1,j} X^j \right)$

$\mathsf{GLWE}_{\vec{S},\sigma}(\Delta M) = \left(A_0 = \sum_{j=0}^{n-1} a_{0,j} X^j, A_1 = \sum_{j=0}^{n-1} a_{1,j} X^j, \ \cdots \ , A_{k-1} = \sum_{j=0}^{n-1} a_{k-1,j} X^j, B = \sum_{j=0}^{n-1} b_j X^j \right)$

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

$E = \sum_{i=0}^{n-1} e_i X^i$

Note that:

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

$= (\Delta m_0 + \Delta m_1 X + \ \cdots \ + \Delta m_{n-1} X^{n-1}) + (e_0 + e_1 X + \ \cdots \ + e_{n-1} X^{n-1})$

$= (\Delta m_0 + e_0) + (\Delta m_1 + e_1) X + \ \cdots \ + (\Delta m_{n-1} + e_{n-1}) X^{n-1}$

Another way to write the formula is:

$$
\begin{aligned}
&B - \sum_{i=0}^{k-1} A_i S_i \\
&= (b_0 + b_1 X + \ \cdots \ + b_{n-1} X^{n-1}) \\
&-(a_{0,0} + a_{0,1} X + \ \cdots \ + a_{0,n-1} X^{n-1})(s_{0,0} + s_{0,1} X + \ \cdots \ + s_{0,n-1} X^{n-1}) \\
&-(a_{1,0} + a_{1,1} X + \ \cdots \ + a_{1,n-1} X^{n-1})(s_{1,0} + s_{1,1} X + \ \cdots \ + s_{1,n-1} X^{n-1}) \\
&- \cdots \\
&-(a_{k-1,0} + a_{k-1,1} X + \ \cdots \ + a_{k-1,n-1} X^{n-1})(s_{k-1,0} + s_{k-1,1} X + \ \cdots \ + s_{k-1,n-1} X^{n-1}) \\
&= \left(b_0 - \left(\sum_{i=0}^{k-1} \sum_{j=0}^{0} (a_{i,0-j} s_{i,j}) - \sum_{i=0}^{k-1} \sum_{j=1}^{n-1} (a_{i,n-j} s_{i,j}) \right) \right) \\
&+ \left(b_1 - \left(\sum_{i=0}^{k-1} \sum_{j=0}^{1} (a_{i,1-j} s_{i,j}) - \sum_{i=0}^{k-1} \sum_{j=2}^{n-1} (a_{i,n+1-j} s_{i,j}) \right) \right) \cdot X \\
&+ \left(b_2 - \left(\sum_{i=0}^{k-1} \sum_{j=0}^{2} (a_{i,2-j} s_{i,j}) - \sum_{i=0}^{k-1} \sum_{j=3}^{n-1} (a_{i,n+2-j} s_{i,j}) \right) \right) \cdot X^2 \\
&\ \cdots \\
&+ \left(b_{n-1} - \left(\sum_{i=0}^{k-1} \sum_{j=0}^{n-1} (a_{i,n-1-j} s_{i,j}) - \sum_{i=0}^{k-1} \sum_{j=n}^{n-1} (a_{i,n+(n-1)-j} s_{i,j}) \right) \right) \cdot X^{n-1} \\
&\text{\# Grouping the terms by same exponents} \\
&= \sum_{h=0}^{n-1} \left(b_h - \left(\sum_{i=0}^{k-1} \sum_{j=0}^{h} (a_{i,h-j} s_{i,j}) - \sum_{i=0}^{k-1} \sum_{j=h+1}^{n-1} (a_{i,n+h-j} s_{i,j}) \right) \right) \cdot X^h \\
&= \sum_{h=0}^{n-1} C_h \cdot X^h, \text{ where } C_h = b_h - \left(\sum_{i=0}^{k-1} \sum_{j=0}^{h} (a_{i,h-j} s_{i,j}) - \sum_{i=0}^{k-1} \sum_{j=h+1}^{n-1} (a_{i,n+h-j} s_{i,j}) \right)
\end{aligned}
$$

In the above $(n-1)$-degree polynomial, notice that each X^h term's coefficient, C_h, can be expressed as an LWE ciphertext ct_h as follows:

$S' = (s_{0,0}, s_{0,1}, \ \cdots \ , s_{0,n-1}, s_{1,0}, s_{1,1}, \ \cdots \ , s_{1,n-1}, \ \cdots \ , s_{k-1,n-1}) = (s'_0, s'_1, \ \cdots \ , s'_{nk-1}) \in \mathbb{Z}_q^{nk}$

$C_h = (a'_0, a'_1, \ \cdots \ , a'_{nk-1}, b_h) \in \mathbb{Z}_q^{nk+1}$

, where $a'_{n \cdot i+j} = \begin{cases} a_{i,h-j} \ (\text{if } 0 \le j \le h) \\ -a_{i,n+h-j} \ (\text{if } h+1 \le j \le n-1) \end{cases}$, b_h is directly obtained from the polynomial B

Note that $b_h - \sum_{i=0}^{nk-1} s'_i a'_i = \Delta m_h + e_h$. This means that C_h can be replaced by its encrypted version, $\mathsf{LWE}_{\vec{s}',\sigma}(\Delta m_h)$, an LWE ciphertext ct_h encrypting the h-th coefficient of M. Therefore, we just extracted $\mathsf{LWE}_{\vec{s}',\sigma}(\Delta m_h)$ from $\mathsf{GLWE}_{\vec{S},\sigma}(\Delta M)$. This operation is called coefficient extraction, which does not add any noise because it simply extracts an LWE ciphertext by reordering the polynomial of the GLWE ciphertext.

Once we have $\mathsf{LWE}_{\vec{s}',\sigma}(\Delta m_h)$, we can key-switch it from $\vec{s}' \to \vec{s}$ (**??**).

⟨Summary ??⟩ GLWE Ciphertext's Coefficient Extraction

Given the following GLWE ciphertext:

$M = \sum_{j=0}^{n-1} m_j X^j \in \mathcal{R}_{\langle n,p \rangle}$

$\vec{S} = \left(S_0 = \sum_{j=0}^{n-1} s_{0,j} X^j, S_1 = \sum_{j=0}^{n-1} s_{1,j} X^j, \cdots, S_{k-1} = \sum_{j=0}^{n-1} s_{k-1,j} X^j \right)$

$\mathsf{GLWE}_{\vec{S},\sigma}(\Delta M) = \left(A_0 = \sum_{j=0}^{n-1} a_{0,j} X^j, A_1 = \sum_{j=0}^{n-1} a_{1,j} X^j, \cdots, A_{k-1} = \sum_{j=0}^{n-1} a_{k-1,j} X^j, B = \sum_{j=0}^{n-1} b_j X^j \right)$

$B = \sum_{i=0}^{k-1} A_i S_i + \Delta M + E \bmod q, \quad E = \sum_{i=0}^{n-1} e_i X^i$

$\mathsf{LWE}_{\vec{s}\prime,\sigma}(\Delta m_h)$ is an LWE ciphertext that encrypts ΔM's h-th coefficient (i.e., Δm_h). $\mathsf{LWE}_{\vec{s}\prime,\sigma}(\Delta m_h)$ can be extracted from $\mathsf{GLWE}_{\vec{S},\sigma}(\Delta M)$ as follows:

$\vec{s}\prime = (s_{0,0}, s_{0,1}, \cdots, s_{0,n-1}, s_{1,0}, s_{1,1}, \cdots, s_{1,n-1}, \cdots, s_{k-1,n-1}) = (s'_0, s'_1, \cdots, s'_{nk-1}) \in \mathbb{Z}_q^{nk}$

$\mathsf{LWE}_{\vec{s}\prime,\sigma}(\Delta m_h) = (a'_0, a'_1, \cdots, a'_{nk-1}, b_h) \in \mathbb{Z}_q^{nk+1}$

, where $a'_{n \cdot i+j} = \begin{cases} a_{i,h-j} \text{ (if } 0 \le j \le h) \\ -a_{i,n+h-j} \text{ (if } h+1 \le j \le n-1) \end{cases}$, b_h is obtained from the polynomial B

Once we have $\mathsf{LWE}_{\vec{s}\prime,\sigma}(\Delta m_h)$, key-switch it from $\vec{s}\prime \to \vec{s}$ (**??**).

D-1.8 Noise Bootstrapping

- Reference: TFHE Deep Dive - Part IV - Programmable Bootstrapping [**?**]

Continuing homomorphic additions of TFHE ciphertexts does not necessarily increase the noise e, because e is randomly generated over the Gaussian distribution, thus adding up many noises would give the mean value of 0. On the other hand, continuing homomorphic multiplications increases the noise, because the noise terms get multiplied, growing its magnitude. Thus, we need to somehow *reset* the noise before it trespasses on the higher bits where plaintext m resides (i.e., preventing the red noise bits from overflowing to the blue plaintext bits as shown in **??**). The process of re-initializing the noise to a smaller value is called noise bootstrapping.

As explained in the beginning of this section, TFHE uses LWE (which is GLWE with $n = 1$) to encrypt & decrypt a plaintext. That is, each plaintext is m (a single number), encoded as a zero-degree polynomial. Further, the secret key S that encrypts each m is a vector $\{s_0, s_1, \cdots, s_{k-1}\}$ instead of a polynomial. On the other hand, TFHE's noise bootstrapping uses homomorphic addition between GLWE ciphertexts and homomorphic multiplication between GLWE and GGSW ciphertexts.

Suppose we have a TFHE ciphertext as follows:

$\mathsf{LWE}_{\vec{s},\sigma}(\Delta m) = (a_0, a_1, \cdots a_{k-1}, b)$

$$b = \sum_{i=0}^{k-1} a_i s_i + \Delta m + e_b$$
$$\vec{s} = (s_0, s_1, \ \cdots \ s_{k-1})$$

, where e_b is a big noise accumulated over a series of many ciphertext (or plaintext) multiplications. The goal of noise bootstrapping is to convert $(a_0, a_1, \ \cdots \ a_{k-1}, b)$ into $(a'_0, a'_1, \ \cdots \ a'_{k-1}, b')$ such that:

$$b' = \sum_{i=0}^{k-1} a'_i s_i + \Delta m + e_s$$

, where e_s is a re-initialized noise.

D-1.8.1 Overview

To implement noise bootstrapping, we create a specially designed $(n-1)$-degree polynomial $V(X)$ called a Lookup Table (LUT). Before explaining $V(X)$, we will first motivate the idea based on a preliminary LUT polynomial $V_q(X)$. Imagine that the polynomial $V_q(X)$'s each degree term X^j has its exponent $j = \Delta m_i + e_*$, a plaintext m_i with some noise $e_* \in \mathbb{Z}_\Delta$, and its corresponding coefficient $v_j = m_i$, which is a noise-free plaintext. Therefore, the $(q-1)$-degree polynomial $V_q(X)$ is defined as follows:

$$\begin{aligned}
V_q(X) &= v_0 + v_1 X^1 + v_2 X^2 + \ \cdots \ + v_{q-1} X^{q-1} \\
&= m_0 X^{\Delta m_0 + e_0} + m_0 X^{\Delta m_0 + e_1} + m_0 X^{\Delta m_0 + e_2} + \ \cdots \ + m_0 X^{\Delta m_0 + e_{\Delta-1}} \ \text{\# total } \Delta \text{ terms} \\
&+ m_1 X^{\Delta m_1 + e_0} + m_1 X^{\Delta m_1 + e_1} + m_1 X^{\Delta m_1 + e_2} + \ \cdots \ + m_1 X^{\Delta m_1 + e_{\Delta-1}} \ \text{\# total } \Delta \text{ terms} \\
&+ \cdots \\
&+ m_{t-1} X^{\Delta m_{t-1} + e_0} + m_{t-1} X^{\Delta m_{t-1} + e_1} + m_{t-1} X^{\Delta m_{t-1} + e_2} + \ \cdots \ + m_{t-1} X^{\Delta m_{t-1} + e_{\Delta-1}} \ \text{\# total } \Delta \text{ terms}
\end{aligned}$$

In the above formula, each m_i and e_k represent every possible plaintext message and error values (where $m_i \in \mathbb{Z}_t$ and $e_k \in \mathbb{Z}_\Delta$).

We design $V_q(X)$ to have the special property that each $v_j X^j$ term represents the special mapping (exponent, coefficient) $= (\Delta m_i + e_*, m_i)$, where e_* can be any value in $\mathbb{Z}_\Delta$. During the TFHE setup stage, we GLWE-encrypt $V_q(X)$ by using our newly defined GLWE key $\vec{S}_{bk}$, a *bootstrapping key*, which is different from the LWE secret key $\vec{s}$. $\vec{S}_{bk}$ is a list of $(n-1)$-degree polynomials with binary coefficients. Later, during the noise bootstrapping stage, we will rotate the coefficients of V by $\Delta m + e$ positions to the left by computing $V \cdot X^{-(\Delta m + e)} = V'$, by using the polynomial coefficient rotation method 1 technique (Summary **??**.1 in **??**). Then, we will extract the polynomial's constant term's coefficient (i.e., the left-most 0-degree term's coefficient in the rotated polynomial V') by using the coefficient extraction technique (**??**). Further, we will encrypt $V_q(X)$ as a GLWE ciphertext at the TFHE setup stage, and thus the rotated $V'_q(X)$'s extracted constant term's coefficient is an LWE encryption of m (i.e., $\mathsf{LWE}_{\vec{s},\sigma}(\Delta m)$) with a re-initialized (i.e., completely reduced) noise.

To summarize, the noise bootstrapping procedure can be conceptually understood (at least for now) as follows:

1. **Input:** $\mathsf{LWE}_{\vec{s},\sigma}(\Delta m + e)$ as a noisy ciphertext encrypting m

2. Convert the input into the form of $X^{-(\Delta m+e)}$ as a rotator of $V_q(X)$ (Lookup Table).
3. Rotate V_q to the left by $\Delta m + e$ positions by computing $V_q \cdot X^{-(\Delta m+e)} = V'_q$.
4. Extract the rotated $V'_q(X)$'s constant term's coefficient m as an LWE encryption, which is $\mathsf{LWE}_{\vec{s},\sigma}(\Delta m)$.
5. **Output:** $\mathsf{LWE}_{\vec{s},\sigma}(\Delta m)$ as an LWE encryption of the plaintext m with a re-initialized noise

The output $\mathsf{LWE}_{\vec{s},\sigma}(\Delta m)$ encrypts the same plaintext message as the input ciphertext, but with completely reduced noise. Therefore, the output $\mathsf{LWE}_{\vec{s},\sigma}(\Delta m)$ can be used for subsequent TFHE homomorphic operations (e.g., addition or multiplication). During this noise bootstrapping process, the polynomial V_q is used as a *dictionary* that contains the mappings from the noisy plaintext $\Delta m + e$ (i.e., as $\Delta m + e = j$ where $v_j X^j$) to the noise-free plaintext m (i.e., as $m = v_j$ where $v_j X^j$). Therefore, V_q is called the Lookup Table (LUT).

Then, what should be the degree of $V_q(X)$? In order for $V_q(X)$ to encode all possible mappings from $\Delta m + e \in \mathbb{Z}_q$ to $m \in \mathbb{Z}_t$, $V_q(X)$ should be a $(q-1)$-degree polynomial. However, q is a very big number, and it is computationally infeasible to manage a $(q-1)$-degree polynomial. Thus, in practice, we instead use a much smaller polynomial $V(X)$ whose degree is only $n-1$. Remember that according to our TFHE setup, $n \ll q$. Therefore, we need a way to *compress* the big ciphertext space $\Delta m + e \in \mathbb{Z}_q$ into a much smaller space $\mathbb{Z}_n$ and encode the compressed values as the exponents of X^j in a *proportionally* correct way. For this proportional compression of $\mathbb{Z}_q \rightarrow \mathbb{Z}_n$, we will use the LWE modulus switching technique learned from **??**.

D-1.8.2 Modulus Switch for Noise Bootstrapping

To avoid using the giant $(q-1)$-degree polynomial V_q, we will compress q possible ciphertext elements $\Delta m + e \in \mathbb{Z}_q$ into n distinct exponents of the $(n-1)$-degree polynomial V, where each $v_j X^j$ term in V represents a mapping from $j \rightarrow v_j$ (i.e., noisy plaintext to noise-free plaintext). However, notice that when we rotate the coefficients of the $(n-1)$-degree polynomial V to the left, as $v_j X^j$ rotates across the boundary between X^0 and X^{n-1} degree terms, v_j's sign flips to $-v_j$ (as shown in the example of **??**). Due to this coefficient sign flip, the $(n-1)$-degree polynomial V can theoretically encode total $2n$ distinct coefficient states as follows: $(v_0, v_1, v_2, \cdots, v_{n-1}, -v_0, -v_1, \cdots, -v_{n-1})$. To move each of these $2n$ distinct coefficients to the constant term's coefficient position in V (i.e., shifting the coefficient v_j to the leftmost term in V), the rotating computation of $V \cdot X^{-j}$ can use $2n$ distinct j values, which are $\{0, 1, 2, \cdots, n-1, n, \cdots, 2n-1\}$, to move each of $(v_0, v_1, v_2, \cdots, v_{n-1}, -v_0, -v_1, \cdots, -v_{n-1})$ coefficients to the constant term's position. This implies that the exponent j in $V \cdot X^{-j}$ can use any of the $2n$ distinct values to cover all possible $2n$ (sign-flipped) coefficient states of V. Also, remember that $j = \Delta m + e$. Therefore, we will switch the modulo of $\Delta m + e$ from $q \rightarrow 2n$. Using the LWE modulus switching technique (**??**), our original LWE ciphertext $\mathsf{LWE}_{\vec{s},\sigma}(\Delta m + e) = (a_0, a_1, \cdots a_{k-1}, b) \in \mathbb{Z}_q^{k+1}$ (i.e., the initial input to the noise bootstrapping procedure) will be converted into the following:

$$\mathsf{LWE}_{\vec{s},\sigma}(\hat{\Delta} m) = (\hat{a}_0, \hat{a}_1, \cdots \hat{a}_{k-1}, \hat{b}) \in \mathbb{Z}_{2n}^{k+1}$$
$$\vec{s} = (s_0, s_1, \cdots s_k) \in \mathbb{Z}_2^k \text{ \# the secret key stays the same, as each } s_i \text{ is binary}$$
$$\hat{\Delta} = \Delta \frac{2n}{q} = \frac{2n}{t} \in \mathbb{Z}_{2n}$$
$$\hat{a}_i = \left\lceil a_i \frac{2n}{q} \right\rfloor \in \mathbb{Z}_{2n}$$

$$\hat{e} = \left\lceil e\frac{2n}{q} \right\rfloor \in \mathbb{Z}_{2n}$$
$$\hat{b} = \left\lceil b\frac{2n}{q} \right\rfloor \approx \sum_{i=0}^{k-1} \hat{a}_i s_i + \hat{\Delta} m + \hat{e} \in \mathbb{Z}_{2n}$$

The degree of $V(X)$ and Security: If our goal were to design the minimal-degree polynomial V whose coefficients map all possible values of the plaintext m, then it would be sufficient to design a t-degree polynomial V. Nonetheless, the reason why we choose the degree of V to be $2n$ instead of t is to guarantee an enough security level– the higher the polynomial degree n is, the safer our scheme is against attacks.

D-1.8.3 Halving the Plaintext Space To be Used

Problematically, the LUT polynomial $V(X)$ rotates *negacyclically*, that is, $V(X) \cdot X^n = -V(X)$ (i.e., coefficients flip their signs with the rotation period of n). More generally:

$$V(X) \cdot X^{-j} = V(X) \cdot X^{2n-j} = V(X) \cdot X^{4n-j} = \cdots$$
$$= V(X) \cdot X^{-(j \bmod 2n)} = \begin{cases} v_j + v_{j+1}X + \cdots, \text{for } 0 \le i < n \\ -v_j - v_{j-1}X - \cdots, \text{for } n \le j < 2n \end{cases}$$

, where v_j denotes the constant term's coefficient after rotating the polynomial $V(X)$ by j positions to the left. Problematically, v_j flips its sign whenever its rotation crosses the boundary between X^0 and X^{n-1}. Given the modulus-switched values $\hat{a}_j$, $\hat{e}$, and $\hat{b}$, we design the following LUT polynomial $V(X)$:

$$\begin{aligned} V(X) &= v_0 + v_1X^1 + v_2X^2 + \cdots + v_{n-1}X^{n-1} \\ &= m_0X^{\hat{\Delta}m_0+\hat{e}_0} + m_0X^{\hat{\Delta}m_0+\hat{e}_1} + m_0X^{\hat{\Delta}m_0+\hat{e}_2} + \cdots + m_0X^{\hat{\Delta}m_0+\hat{e}_{\hat{\Delta}-1}} \text{ \# total } \hat{\Delta} \text{ terms} \\ &+ m_1X^{\hat{\Delta}m_1+\hat{e}_0} + m_1X^{\hat{\Delta}m_1+\hat{e}_1} + m_1X^{\hat{\Delta}m_1+\hat{e}_2} + \cdots + m_1X^{\hat{\Delta}m_1+\hat{e}_{\hat{\Delta}-1}} \text{ \# total } \hat{\Delta} \text{ terms} \\ &+ \cdots \\ &+ m_{t/2-1}X^{\hat{\Delta}m_{t/2-1}+\hat{e}_0} + m_{t/2-1}X^{\hat{\Delta}m_{t/2-1}+\hat{e}_1} + m_{t/2-1}X^{\hat{\Delta}m_{t/2-1}+\hat{e}_2} + \cdots + m_{t/2-1}X^{\hat{\Delta}m_{t/2-1}+\hat{e}_{\hat{\Delta}-1}} \\ &\text{\# total } \hat{\Delta} \text{ terms} \end{aligned}$$

Remember that by computing $V(X) \cdot X^{-j}$ for $j = \{0, 1, \cdots, n-1\}$, we can rotate $V(X)$'s coefficients to the left by $\{0, 1, \cdots n-1\}$ positions. For each j-slot rotation of $V(X)$ where $j = \{0, 1, \cdots, n-1\}$, the rotated polynomial $V'(X)$ gets the following values as the constant-term's coefficient:

$$\overbrace{\underbrace{\Delta m_0, \Delta m_0, \cdots}_{\text{coeff. of } X^{\hat{\Delta}m_0+\hat{e}_*}} \underbrace{\Delta m_1, \Delta m_1, \cdots}_{\text{coeff. of } X^{\hat{\Delta}m_1+\hat{e}_*}} \cdots \underbrace{\Delta m_{t/2-1}, \Delta m_{t/2-1}, \cdots}_{\text{coeff. of } X^{\hat{\Delta}m_{t/2-1}+\hat{e}_*}}}^{V'(X)\text{'s constant term's coefficient for } j=\{0,1,\cdots n-1\} \text{ rotations}}$$
($\hat{\Delta}$ repetitions for each group)

In the above expression, $\hat{e}_*$ is a noise that can range from $[0, \hat{\Delta})$. Note that all of $\hat{\Delta}m_i + \hat{e}_0, \hat{\Delta}m_i + \hat{e}_1, \cdots, \hat{\Delta}m_i + \hat{e}_{\hat{\Delta}-1}$ exponents are designed to be mapped to the same coefficient value, m_i, which aligns with the fact that their underlying plaintext m_i is the same when decrypted (once their associated noise $\hat{e}_*$ gets eliminated). This is why each m_i is redundantly used $\hat{\Delta}$ times in a row as coefficients in $V(X)$. We can view this sequential repetition of coefficients as having a robustness of mapping each $\hat{\Delta}m_i + \hat{e}_*$ to m_i against any noise $\hat{e}_* \in \mathbb{Z}_{\hat{\Delta}}$.

So far, the above sequence of $m_0, m_1, \cdots, m_{t/2-1}$ coefficients is what we expect $V'(X)$ (i.e., the rotated polynomial) to return as its constant term's coefficient for each of $0, 1, \cdots, n-1$ rotations (where each $m_i + 1 = m_{i+1}$). However, the correctness of the coefficient mappings breaks when the rotation count is between $[n, 2n-1)$, because their coefficients flip their signs when they cross the term's boundary from X^0 to X^{n-1}, due to the polynomial ring's negacyclic nature. Specifically, the constant term's coefficient values of the rotated polynomial $V'(X)$ are as follows for each rotation of $n, n+1, \cdots, 2n-1$ positions:

$$\overbrace{\underbrace{-m_0, -m_0, \cdots}_{\hat{\Delta} \text{ repetitions}} \underbrace{-m_1, -m_1, \cdots}_{\hat{\Delta} \text{ repetitions}} \cdots \underbrace{-m_{t/2-1}, -m_{t/2-1}, \cdots}_{\hat{\Delta} \text{ repetitions}}}^{V'(X)\text{'s constant term's coefficient after each of } n, n+1, \cdots 2n-1 \text{ rotations}}$$

$$\underbrace{}_{-\text{coeff. of } X^{\hat{\Delta} m_0 + \hat{e}_*}} \underbrace{}_{-\text{coeff. of } X^{\hat{\Delta} m_1 + \hat{e}_*}} \qquad \underbrace{\phantom{-m_{t/2-1}, -m_{t/2-1}, \cdots}}_{-\text{coeff. of } X^{\hat{\Delta} m_{t/2-1} + \hat{e}_*}}$$

As we can see above, the rotated $V'(X)$'s constant term's coefficient shows a negacyclic pattern with the rotation period of n, where the second n-rotation group's coefficients are exactly the negated values of the first n-rotation group's values. Let's understand why this negacyclic behavior breaks the (exponent, coefficient) $= (\hat{\Delta} m + \hat{e}, m)$ mappings. Since TFHE's plaintext and ciphertext values are defined in rings, as we rotate the LUT polynomial $V(X)$, we ideally want the rotated polynomial $V'(X)$'s constant term's value (i.e., mapped plaintext value) to wrap around in a circular manner, representing a ring pattern (with sequential $\hat{\Delta}$ repetitions of each value to be resistant against up to a $\hat{e}_* \in Z_{\hat{\Delta}}$ noise). However, the negacyclic nature of a polynomial ring makes the constant term's value of the second-half rotation group problematic, because they are exact negations of those of the first-half rotation group, breaking the circular wrapping-around ring pattern between the first-group values and the second-group values.

To summarize the problem, $V(X)$ has a limitation in becoming a perfect LUT, because it preserves the correct mappings of (exponent, coefficient) $= (\hat{\Delta} m + \hat{e}, m)$ only for one half of $i \in \mathbb{Z}_t$, not for the other half.

Solution: Good news is that we have observed that $V(X)$'s mappings of (exponent, coefficient) $= (\hat{\Delta} m + \hat{e}, \Delta m)$ preserve their ring-pattern consistency if $V(X)$ is rotated no more than $n-1$ positions (i.e., the first-half rotation group). Therefore, the easiest solution to avoid the negacyclic problem of the LUT polynomial rotation is that the application of TFHE restricts $V(X)$ to be rotated no more than $n-1$ positions *by design* during the noise bootstrapping. To enforce this, when the TFHE application's computation pipeline processes plaintext values (in its original plaintext computation logic before considering any homomorphic operations), the application should ensure to involve only some pre-defined continuous $\frac{t}{2}$ modulo values within $\mathbb{Z}_t$ as the possible inputs and outputs of each computation step. This constraint effectively ensures the possible values of $\hat{\Delta} m + \hat{e}$ to be continuous n values within $\mathbb{Z}_{2n}$. Since the LUT polynomial $V(X)$ gets rotated by computing $V(X) \cdot X^{-(\hat{\Delta} m + \hat{e})}$, as the application restricts $\hat{\Delta} m + \hat{e}$ to be at most $n-1$ (out of $2n-1$) by its application design, $V(X)$ will be rotated at most $n-1$ positions during the noise bootstrapping. Thus, we can prevent the occurrences of the problematic $\{n, n+1, \cdots, 2n-1\}$ rotations that flip the signs of coefficients.

To summarize, at the cost of halving the application's usable plaintext values to some continuous $\frac{t}{2}$ values within $\mathbb{Z}_t$, we can prevent $V(X)$'s negacyclic rotation problem, and thereby preserve

$V(X)$'s correct mappings of (exponent, coefficient) $= (\hat{\Delta}m + \hat{e}, m)$.

Considering all these, our final LUT polynomial $V(X)$ is as follows:

⟨Summary ??⟩ Structure of Lookup Table Polynomial $V(X)$

$$
\begin{aligned}
V(X) &= v_0 + v_1X^1 + v_2X^2 + \ \cdots\ + v_{n-1}X^{n-1} \\
&= m_0X^{\hat{\Delta}m_0+\hat{e}_0} + m_0X^{\hat{\Delta}m_0+\hat{e}_1} + m_0X^{\hat{\Delta}m_0+\hat{e}_2} + \ \cdots\ + m_0X^{\hat{\Delta}m_0+\hat{e}_{\hat{\Delta}-1}} \\
&+ m_1X^{\hat{\Delta}m_1+\hat{e}_0} + m_1X^{\hat{\Delta}m_1+\hat{e}_1} + m_1X^{\hat{\Delta}m_1+\hat{e}_2} + \ \cdots\ + m_1X^{\hat{\Delta}m_1+\hat{e}_{\hat{\Delta}-1}} \\
&+ \cdots \\
&+ m_{t/2-1}X^{\hat{\Delta}m_{t/2-1}+\hat{e}_0} + m_{t/2-1}X^{\hat{\Delta}m_{t/2-1}+\hat{e}_1} + m_{t/2-1}X^{\hat{\Delta}m_{t/2-1}+\hat{e}_2} \\
&\quad + \cdots\ + m_{t/2-1}X^{\hat{\Delta}m_{t/2-1}+\hat{e}_{\hat{\Delta}-1}}
\end{aligned}
$$

, where $\hat{\Delta} = \Delta \cdot \frac{2n}{q} = \frac{2n}{t}$. The application should ensure that $m_0, m_1, \cdots, m_{t/2-1}$ are some continuous modulo-$\frac{t}{2}$ values in $\mathbb{Z}_t$. This constraint ensures $V(X)$'s rotation positions $\hat{\Delta}m_i + \hat{e}_*$ (where $e_* \in \mathbb{Z}_{\hat{\Delta}}$) to be most n continuous possibilities, preventing $V(X)$ from making more than 1 full-cycle rotation that triggers a negacyclic problem.

D-1.8.4 Blind Rotation

Blind rotation refers to rotating an *encrypted* polynomial's coefficients so that it is not possible to know how many positions the polynomial's coefficients are rotated, and after the rotation, it is not possible to see which coefficient has moved to which degree term. Blind rotation uses the basic polynomial rotation method 1 technique (Summary **??** in **??**), with the difference that the $V(X) \cdot X^{-i}$ computation is done homomorphically.

Note that $X^{\hat{\Delta}m+\hat{e}_b} = X^{\hat{b}-\sum_{i=0}^{k-1}\hat{a}_is_i}$, so we can rotate V by computing $V \cdot X^{-(\hat{b}-\sum_{i=0}^{k-1}\hat{a}_is_i)} = V \cdot X^{-\hat{b}+\sum_{i=0}^{k-1}\hat{a}_is_i}$. In fact, we cannot directly compute the LWE decryption formula $-\hat{b}+\sum_{i=0}^{k-1}\hat{a}_is_i$ (or $\hat{b}-\sum_{i=0}^{k-1}\hat{a}_is_i$) without the knowledge of the LWE secret key S. Nevertheless, there is a mathematical work-around to compute $V \cdot X^{-\hat{b}+\sum_{i=0}^{k-1}\hat{a}_is_i}$ without the knowledge of the secret key S, provided we are given $\{GGSW^{\beta,l}_{\vec{S}_{bk},\sigma}(s_i)\}_{i=0}^{k-1}$ at the TFHE setup stage. Note that $\{GGSW^{\beta,l}_{\vec{S}_{bk},\sigma}(s_i)\}_{i=0}^{k-1}$ is a GGSW encryption of the LWE secret key S, encrypted by the GLWE secret key $\vec{S}_{bk}$ (i.e., a *bootstrapping* key). We use $\vec{S}_{bk}$ to homomorphically compute $V \cdot X^{-\hat{b}+\sum_{i=0}^{k-1}\hat{a}_is_i}$ (i.e., blindly rotate the coefficients of V to the left by $\hat{b} + \sum_{i=0}^{k-1}\hat{a}_is_i$ positions), according to the following procedure:

1. GLWE-encrypt the polynomial V with the bootstrapping key $\vec{S}_{bk}$ at the TFHE setup stage, so that each coefficient of V gets encrypted.
2. Compute $V_0 = V \cdot X^{-\hat{b}}$, which is basically rotating V's polynomials by $\hat{b}$ positions to the left. Since $\hat{b}$ is a known value visible in the LWE ciphertext, we can directly compute the rotation of $V_0 = V \cdot X^{-\hat{b}}$.
3. Compute $V_1 = V_0 \cdot X^{\hat{a}_0s_0} = s_0 \cdot (V_0 \cdot X^{\hat{a}_0} - V_0) + V_0$. This formula works for the special case where $s_0 \in \{0, 1\}$: if $s_0 = 0$, then $V_1 = V_0$; else if $s_0 = 1$, then $V_1 = V_0 \cdot X^{\hat{a}_0}$. Computing $s_0 \cdot (V_0 \cdot X^{\hat{a}_0} - V_0) + V_0$ is done as a TFHE homomorphic addition and multiplication. We call this blind rotation of V_0, where the selection bit s_0 (i.e., the 1st element of the secret key vector S) is encrypted as a GGSW ciphertext by using $\vec{S}_{bk}$ (i.e. the bootstrapping key) and

V_0 is an encrypted polynomial as a GLWE ciphertext. Multiplying GLWE-encrypted V_0 with $x^{\hat{a}_0}$ is done by GLWE ciphertext-to-plaintext multiplication (**??**), and subtracting the result by GLWE-encrypted V_0 is done by GLWE ciphertext-to-ciphertext addition/subtraction (**??**), and multiplying the result by GGSW-encrypted s_0 is done by GLWE-to-GGSW multiplication (**??**), and adding the result with GLWE-encrypted V_0 is done by GLWE ciphertext-to-ciphertext addition. If $s_0 = 1$, then the formula's $X^{\hat{a}_0}$ term gets multiplied to V_0, which effectively rotates V_0's coefficients by $\hat{a}_0$ positions to the right. Else if $s_0 = 0$, then V_0 does not get rotated and stays the same. In both cases, the resulting V_1 is encrypted as a new GLWE ciphertext. During this blind rotation, unless we have the knowledge of s_0 and $\vec{S}_{bk}$, it is impossible to know whether V_0 has been rotated or not, and also how many positions have been rotated.

4. By using the same blind rotation method as in the previous step, compute the GLWE encryption of $V_2 = V_1 \cdot X^{\hat{a}_1 s_1} = s_1 \cdot (V_1 \cdot X_1^{\hat{a}} - V_1) + V_1$. Note that we have the following publicly known components: $\hat{a}_1$, a GLWE encryption of V_1, and a GGSW ciphertext of s_1 encrypted by using $\vec{S}_{bk}$.
5. Continue to compute the GLWE encryption of $V_3, V_4, \cdots, V_k$ in the same manner, and we finally get a GLWE encryption of V_k, whose computed value is equivalent to:
$$\begin{aligned} V' &= V_k \\ &= V_{k-1} \cdot X^{\hat{a}_{k-1} s_{k-1}} \\ &= V_0 \cdot X^{\hat{a}_0 s_0} X^{\hat{a}_1 s_1} \cdots X^{\hat{a}_{k-1} s_{k-1}} \\ &= V \cdot X^{-\hat{b} + \sum_{i=0}^{k-1} \hat{a}_i s_i} \\ &= V \cdot X^{-(\hat{\Delta} m + \hat{e}_b)} \end{aligned}$$

This means that the GLWE encryption of the final polynomial V_k will have the coefficient m in its constant term, as $V(X)$ is designed to have the mapping $(\hat{\Delta} m + \hat{e}_*, m)$.

Note that while we restrict the application's plaintext space usage to some continuous $\frac{t}{2}$ modulo values within $\mathbb{Z}_t$ (**??**), this restriction does not exist in the ciphertext space. That is, it is okay for blind rotations to rotate $V(X)$ more than n positions during the intermediate steps, because their invalid positions can be brought back to valid ones by their subsequent steps. Therefore, what matters for the noise bootstrapping correctness is only the completed state $V_k(X)$. The rotation-completed $V_k(X)$ must have been rotated $\hat{\Delta} m + \hat{e}$ positions to the left. Therefore, we only need to ensure that $\hat{\Delta} m + \hat{e}$ falls within our pre-defined continuous $\frac{t}{2}$ modulo range within the $\mathbb{Z}_t$ domain, which is equivalent to ensuring that the aggregate rotation count is at most $n-1$ positions to avoid coefficient extraction of any double-signed contradicting coefficients.

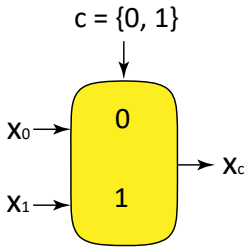


Figure 13: An illustration of the MUX logic gate

Homomorphic MUX Logic Gate: In step 3, the formula $s_0 \cdot (V_0 \cdot X^{\hat{a}_0} - V_0) + V_0$ implements the MUX logic gate as shown in **??**, where in our case s_0 is the selection bit that chooses between the two inputs: V_0 and $V_0 \cdot X^{\hat{a}_0}$. If s_0 is 1, then the output is $V_0 \cdot X^{\hat{a}_0}$; otherwise, the output is V_0.

In our design, the homomorphic computation of $s_0 \cdot (V_0 \cdot X^{\hat{a}_0} - V_0) + V_0$ effectively implements a homomorphic MUX gate, where the two inputs are LWE-encrypted and the selection bit is GGSW-encrypted.

D-1.8.5 Coefficient Extraction

Finally, we use the coefficient extraction technique (**??**) to extract the rotated polynomial $V'(X)$'s constant term's coefficient as an encryption of m: $\mathsf{LWE}_{\vec{s},\sigma}(\Delta m)$. At this point, the original LWE ciphertext's old noise $\hat{e}_b$ has gone, and the bootstrapped new ciphertext $\mathsf{LWE}_{\vec{s},\sigma}(\Delta m)$ has a newly generated small noise e_s.

In fact, the homomorphic MUX logic in the blind rotation procedure (**??**) involves lots of ciphertext multiplications and additions, which can accumulate additional noise until we reach the point of coefficient extraction. In order to limit the accumulating noise during the series of MUX logic, we can carefully adjust the security parameters. For example, we can design a narrower Gaussian distribution for sampling the noise e, while designing a sufficiently large n to compensate for the reduced noise. Meanwhile, increasing q rather makes the system less secure, because it becomes more vulnerable to lattice reduction attacks.

D-1.8.6 Noise Bootstrapping Summary

TFHE's noise bootstrapping procedure is summarized as follows:

⟨Summary ??⟩ TFHE Noise Bootstrapping Procedure

Lookup Table Encryption: Encrypt the LUT polynomial $V(X)$ as a GLWE ciphertext by using the bootstrapping key $\vec{S}_{bk}$.

1. **Modulus Switch:** Change the modulus of the TFHE ciphertext $\mathsf{LWE}_{\vec{s},\sigma}(\Delta m + e_b)$ from $q \to 2n$ to get $\mathsf{LWE}_{\vec{s},\sigma}(\hat{\Delta} m + \hat{e}_b)$, where $\hat{\Delta} = \Delta \cdot \dfrac{2n}{q} = \dfrac{2n}{t}$.

2. **Blind Rotation:** Rotate the GLWE-encrypted polynomial V by $(\hat{b} - \sum_{i=0}^{k-1} \hat{a}_i s_i) = (\hat{\Delta} m + \hat{e}_b)$ positions to the left, by recursively computing:
   $V_0 = V \cdot X^{-\hat{b}}$
   $V_1 = V_0 \cdot X^{\hat{a}_0 s_0} = s_0 \cdot (V_0 \cdot X^{\hat{a}_0} - V_0) + V_0$
   $\vdots$
   $V_k = V_{k-1} \cdot X^{\hat{a}_{k-1} s_{k-1}} = s_{k-1} \cdot (V_{k-1} \cdot X^{\hat{a}_0} - V_0) + V_{k-1}$
   $= V_0 \cdot X^{\hat{a}_0 s_0} X^{\hat{a}_1 s_1} \cdots X^{\hat{a}_{k-1} s_{k-1}}$
   $= V \cdot X^{-\hat{b} + \sum_{i=0}^{k-1} \hat{a}_i s_i}$
   $= V \cdot X^{-(\hat{\Delta} m + \hat{e}_b)}$

   Each step of the actual blind rotation above is computed as the following TFHE ciphertext-to-ciphertext multiplication and addition:
   $\mathsf{GLWE}_{\vec{S},\sigma}(V_{i+1}) = \mathsf{GGSW}^{\beta,l}_{\vec{S},\sigma}(s_i) \cdot (\mathsf{GLWE}_{\vec{S},\sigma}(V_i) \cdot X^{a_0} - \mathsf{GLWE}_{\vec{S},\sigma}(V_i)) + \mathsf{GLWE}_{\vec{S},\sigma}(V_i)$

3. **Coefficient Extraction:** Homomorphically extract the constant term's coefficient m from the rotated polynomial V_k, which is $\mathsf{LWE}_{\vec{s},\sigma}(\hat{\Delta}m)$.

Halving the Usable Plaintext Range: Problematically, the LUT polynomial V rotates negacyclically. To avoid this problem, we require the application to ensure that the plaintext m uses only continuous $\frac{t}{2}$ modulo values within $\mathbb{Z}_t$. This way, we avoid rotating $V(X)$ more than $n-1$ positions that cause coefficient extraction of double-signed contradicting coefficients.

D-1.8.7 Example: Noise Bootstrapping

Suppose the GLWE security setup: $n = 16$, $t = 8$, $q = 64$, $k = 8$

$\mathbb{Z}_{t=8} = \{-4, -3, -2, -1, 0, 1, 2, 3\}$

$\mathbb{Z}_{q=64} = \{-32, -31, -30, \cdots, 29, 30, 31\}$

$\Delta = \frac{q}{t} = \frac{64}{8} = 8$

And suppose we have the following LWE ciphertext:

$\vec{s} = (1, 0, 0, 1, 1, 1, 0, 1) = \mathbb{Z}_2^{k=8}$

$m = 1 \in \mathbb{Z}_{t=8}$

$\Delta m = 1 \cdot 8 = 8 \in \mathbb{Z}_{q=64}$

$\mathsf{LWE}_{\vec{s},\sigma}(\Delta m) = (a_0, a_1, a_2, a_3, a_4, a_5, a_6, a_7, b) = (8, -28, 4, -32, 0, 31, -6, 7, 24) \in \mathbb{Z}_{q=64}^{k+1=9}$

$e = 2 \in \mathbb{Z}_{q=64}$ (should be the case that $|e| < \frac{\Delta}{2} = 4$ for correct decryption)

$b = \sum_{i=0}^{7} a_i s_i + \Delta m + e = (8 - 32 + 31 + 7) + 8 + 2 = 24 \in \mathbb{Z}_{q=64}$

Now then, the TFHE noise bootstrapping procedure is as follows:

1. **Modulus Switch:** Switch the modulus of $\mathsf{LWE}_{\vec{s},\sigma}(\Delta m)$ From $q \to 2n$, which is from $64 \to 32$. After the modulus switch, the original LWE ciphertext is converted as follows:

   $\mathbb{Z}_{2n=32} = \{-16, -15, -14, \cdots, 13, 14, 15\}$

   $\vec{s} = (1, 0, 0, 1, 1, 1, 0, 1) = \mathbb{Z}_2^{k=8}$

   $\hat{\Delta} = \Delta\frac{2n}{64} = 8\frac{32}{64} = 4$

   $\hat{\Delta}m = 4 \cdot 1 = 4 \in \mathbb{Z}_{2n=32}$

   $\hat{e} = \left\lceil e\frac{2n}{q} \right\rfloor = \left\lceil 2\frac{32}{64} \right\rfloor = 1 \in \mathbb{Z}_{2n=32}$

   $\mathsf{LWE}_{\vec{s},\sigma}(\hat{\Delta}m) = (\hat{a}_0, \hat{a}_1, \hat{a}_2, \hat{a}_3, \hat{a}_4, \hat{a}_5, \hat{a}_6, \hat{a}_7, \hat{b}) \in \mathbb{Z}_{2n=32}^{k+1=9}$

   $= (\left\lceil 8\frac{32}{64} \right\rfloor, \left\lceil -28\frac{32}{64} \right\rfloor, \left\lceil 4\frac{32}{64} \right\rfloor, \left\lceil -32\frac{32}{64} \right\rfloor, \left\lceil 0\frac{32}{64} \right\rfloor, \left\lceil 31\frac{32}{64} \right\rfloor, \left\lceil -6\frac{32}{64} \right\rfloor, \left\lceil 7\frac{32}{64} \right\rfloor, \left\lceil 24\frac{32}{64} \right\rfloor)$

   $= (4, -14, 2, -16, 0, 16, -3, 4, 12)$

   Note that $\sum_{i=0}^{7} (\hat{a}_i s_i) + \hat{\Delta}m + \hat{e} = (4 - 16 + 16 + 4) + 4 + 1 = 13 \in \mathbb{Z}_{2n=32}$

$\hat{b} = 12 \approx 13 = \sum_{i=0}^{7} (\hat{a}_i s_i) + \hat{\Delta} m + \hat{e}$

This small difference in $\hat{b}$ comes from the aggregated noises of rounding $\hat{a}_0, \hat{a}_1, \cdots, \hat{e}$ during the modulus switch.

2. **Blind Rotation:** We assume that the application avoids the problem of negacyclic polynomial rotation by ensuring that the usable plaintext values are the following continuous $\frac{8}{2}$ modulo values within $\mathbb{Z}_8 = \{-4, -3, -2, -1, 0, 1, 2, 3\}$, which are $\{-2, -1, 0, 1\}$. This implies that the only possible values of $i = \hat{\Delta} m + \hat{e}$ in $V(X) \cdot X^i$ will be: $i = \{-8, -7, \cdots, 6, 7\}$. Based on these requirements, **??** is the Lookup Table polynomial $V(X)$ that maps $\hat{\Delta} m + \hat{e}$ to Δm.

$$
\begin{aligned}
V(X) &= v_0 + v_1 X + v_2 X^2 + v_3 X^3 + v_4 X^4 + v_5 X^5 + v_6 X^6 + v_7 X^7 \\
&\quad + v_8 X^8 + v_9 X^9 + v_{10} X^{10} + v_{11} X^{11} + v_{12} X^{12} + v_{13} X^{13} + v_{14} X^{14} + v_{15} X^{15} \\
&= 0 + 0X + 0X^2 + 0X^3 + 1X^4 + 1X^5 + 1X^6 + 1X^7 \\
&\quad + 2X^8 + 2X^9 + 2X^{10} + 2X^{11} + 1X^{12} + 1X^{13} + 1X^{14} + 1X^{15}
\end{aligned}
$$

$i = \hat{\Delta} m + \hat{e}$ (in $V \cdot X^{-i}$)	-8 (11000_2)	-7 (11001_2)	-6 (11010_2)	-5 (11011_2)	-4 (11100_2)	-3 (11101_2)	-2 (11110_2)	-1 (11111_2)
constant term's coeff. of $V \cdot X^{-i}$	-2 110_2	-2 110_2	-2 110_2	-2 110_2	-1 111_2	-1 111_2	-1 111_2	-1 11100_2
m **(plaintext)**	-2	-2	-2	-2	-1	-1	-1	-1

$i = \hat{\Delta} m + \hat{e}$ (in $V \cdot X^{-i}$)	0 (000_2)	1 (000_2)	2 (000_2)	3 (000_2)	4 (001_2)	5 (001_2)	6 (001_2)	7 (001_2)
constant term's coeff. of $V \cdot X^{-i}$	0 00000_2	0 00000_2	0 00000_2	0 00000_2	1 00100_2	1 00100_2	1 00100_2	1 00100_2
m **(plaintext)**	0	0	0	0	1	1	1	1

Table 4: The Lookup Table for $n = 16, q = 64, t = 8$ LWE setup. Orange is the plaintext m's bits. Green is the noise e's bits.

Note that $V(X)$'s coefficients for the $X^8 \sim X^{15}$ terms are $\{2, 1\}$ instead of $\{-2, -1\}$, so that if V gets rotated by $\{-8, -7, -6, -5, 4, 5, 6, 7\}$ slots to the left, the constant term's coefficient flips its sign to $\{-2, -1\}$ due to wrapping around the boundary of the n exponent.
During the actual bootstrapping, we will do a blind rotation of **??**'s $V(X)$ (which is GLWE-encrypted) by $\hat{b} - \sum_{i=0}^{7} \hat{a}_i s_i = 4$ positions to the left, which is computed as follows:

$\hat{\Delta} m + \hat{e} = \hat{b} - \sum_{i=0}^{7} \hat{a}_i s_i = 12 - (4 - 16 + 16 + 4) = 4 \bmod 32 \in \mathbb{Z}_{2n=32}$

In **??**, if the rotation count $i = 4$, the corresponding constant term's coefficient is $v_4 = 1 = m$. As $\Delta = 4$, we finally get $\mathsf{LWE}_{\vec{s},\sigma}(\Delta m) = 1$.

The actual blind rotation is computed as follows:
$\vec{s} = (1, 0, 0, 1, 1, 1, 0, 1)$
$\mathsf{LWE}_{\vec{s},\sigma}(\hat{\Delta} m) = (\hat{a}_0, \hat{a}_1, \hat{a}_2, \hat{a}_3, \hat{a}_4, \hat{a}_5, \hat{a}_6, \hat{a}_7, \hat{b}) = (4, -14, 2, -16, 0, 16, -3, 4, 12)$
$V_0 = V \cdot X^{-\hat{b}} = V \cdot X^{-12} = v_{12} + v_{13} X + v_{14} X^2 + \cdots$
$V_1 = V_0 \cdot X^{\hat{a}_0 s_0} = s_0 \cdot (V_0 \cdot X^{\hat{a}_0} - V_0) + V_0 = V_0 \cdot X^4 = v_8 + v_9 X + v_{10} X^2 + \cdots$
$V_2 = V_1 \cdot X^{\hat{a}_1 s_1} = s_1 \cdot (V_1 \cdot X^{\hat{a}_1} - V_1) + V_1 = V_1 = v_8 + v_9 X + v_{10} X^2 + \cdots$
$V_3 = V_2 \cdot X^{\hat{a}_2 s_2} = s_2 \cdot (V_2 \cdot X^{\hat{a}_2} - V_2) + V_2 = V_2 = v_8 + v_9 X + v_{10} X^2 + \cdots$
$V_4 = V_3 \cdot X^{\hat{a}_3 s_3} = s_3 \cdot (V_3 \cdot X^{\hat{a}_3} - V_3) + V_3 = V_3 \cdot X^{-16} = -v_8 - v_9 X - v_{10} X^2 - \cdots$
$V_5 = V_4 \cdot X^{\hat{a}_4 s_4} = s_4 \cdot (V_4 \cdot X^{\hat{a}_4} - V_4) + V_4 = V_4 \cdot X^0 = -v_8 - v_9 X - v_{10} X^2 - \cdots$

$V_6 = V_5 \cdot X^{\hat{a}_5 s_5} = s_5 \cdot (V_5 \cdot X^{\hat{a}_5} - V_5) + V_5 = V_5 \cdot X^{16} = v_8 + v_9 X + v_{10} X^2 + \cdots$
$V_7 = V_6 \cdot X^{\hat{a}_6 s_6} = s_6 \cdot (V_6 \cdot X^{\hat{a}_6} - V_6) + V_6 = V_6 = v_8 + v_9 X + v_{10} X^2 + \cdots$
$V_8 = V_7 \cdot X^{\hat{a}_7 s_7} = s_7 \cdot (V_7 \cdot X^{\hat{a}_7} - V_7) + V_7 = V_7 \cdot X^4 = v_4 + v_5 X + v_6 X^2 + \cdots$

The final output of blind rotation is the GLWE ciphertext of V_8, $\mathsf{GLWE}_{\vec{S},\sigma}(V_8)$, whose constant term's coefficient is $v_4 = m = 1$.

Each step of the actual blind rotation above is computed as the following TFHE ciphertext-to-ciphertext multiplication:
$\mathsf{GLWE}_{\vec{S},\sigma}(V_{i+1}) = \mathsf{GGSW}^{\beta,l}_{\vec{S},\sigma}(s_i) \cdot (\mathsf{GLWE}_{\vec{S},\sigma}(V_i) \cdot X^{a_0} - \mathsf{GLWE}_{\vec{S},\sigma}(V_i)) + \mathsf{GLWE}_{\vec{S},\sigma}(V_i)$

We will leave this computation for the reader's exercise.

3. **Coefficient Extraction:** At the end of blind rotation, we finally get the following GLWE ciphertext:
$\mathsf{GLWE}_{\vec{S},\sigma}(V_8)$
$= \mathsf{GLWE}_{\vec{S},\sigma}(\hat{\Delta} \cdot (v_4 + v_5 X + v_6 X^2 + v_7 X^3 + v_8 X^4 + v_9 X^5 + v_{10} X^6 + v_{11} X^7 + v_{12} X^8 + v_{13} X^9 + v_{14} X^{10} + v_{15} X^{11} - v_0 X^{12} - v_1 X^{13} - v_2 X^{14} - v_3 X^{15}))$
$= \mathsf{GLWE}_{\vec{S},\sigma}(\hat{\Delta} \cdot (1 + 1X + 1X^2 + 1X^3 + 2X^4 + 2X^5 + 2X^6 + 2X^7 + 1X^8 + 1X^9 + 1X^{10} + 1X^{11} - 0X^{12} - 0X^{13} - 0X^{14} - 0X^{15}))$
$= \left(A_0 = \sum_{j=0}^{15} (a_{0,0} + a_{0,1} X + \cdots), A_1 = \cdots, A_{k-1} = \cdots, B = \sum_{j=0}^{15} b_j X^j \right)$
Now, we extract the constant term's coefficient of the encrypted polynomial $\mathsf{GLWE}_{\vec{S},\sigma}(\hat{\Delta} \cdot (1 + 1X + 1X^2 + \cdots))$ by using the coefficient extraction formula (Summary **??**). Specifically, we will extract the constant term's coefficient, which corresponds to $\mathsf{LWE}_{\vec{s},\sigma}(\Delta m_0)$. We extract $\mathsf{LWE}_{\vec{s},\sigma}(\Delta m_0)$ by computing the following:
$\mathsf{LWE}_{\vec{s},\sigma}(\Delta m_0) = (a'_0, a'_1, \cdots, a'_k, b_h)$

, where $a'_{n \cdot i + j} = \begin{cases} a_{i,0-j} \text{ (if } 0 \le j \le 0) \\ a_{i,n+0-j} \text{ (if } 0+1 \le j \le n-1) \end{cases}$, b_0 is obtained from the polynomial B

D-1.8.8 Discussion

- **Programmable Bootstrapping**: While the bootstrapping (**??**) uses a simple Lookup Table $V(X)$ which maps $\Delta m + e$ to Δm, we can edit the coefficients of $V(X)$ to make $\Delta m + e$ map to different values. For example, an altered mappings between the inputs and outputs to LUT can implement logic gates such as AND, OR, XOR, CMUX, etc, which will be explained in **??**. Such edited mappings between the exponents and coefficients in $V(X)$ are called programmable bootstrapping. If we encrypt $V(X)$ as a GLWE ciphertext, we can hide the mappings as well as each input instance, which effectively implements *functional encryption*. Note that both the vanilla bootstrapping (**??**) and programmable bootstrapping (**??**) generate the same amount of noise.
- **Bootstrapping Noise**: During the bootstrapping's LUT polynomial $V(X)$ rotation, we perform many TFHE multiplications in the homomorphic MUX gates to derive $V_0 \cdots V_k$, which inevitably creates additional noises before the the noise gets re-initialized at the end. However,

a careful parameter choice can limit the growth of this additional noise during modulus switch and blind rotation.

D-1.8.9 Application: Gate Bootstrapping

Besides implementing the homomorphic MUX logic gate used during blind rotation (**??**), it is possible to leverage the LUT polynomial $V(X)$ to implement other homomorphic logic gates such as AND, NAND, OR, XOR, etc. When implementing these gates, each ciphertext is an encryption of a single-bit plaintext (or several bits can be bundled up in a linear combination formula and be processed simultaneously by using LUT). Suppose $q = 32$, $t = 8$, $m \in \mathbb{Z}_8 = \{-4, -3, -2, -1, 0, 1, 2, 3\}$, $\Delta = \frac{q}{t} = 4$, $\hat{\Delta} = \frac{\Delta \cdot 2n}{q} = 2$, and we encode the gate input into LWE plaintext as $0 \to -1$, and $1 \to 1$, and the maximum (accumulated) noise $e = [-1, 1]$.

Lookup Table Polynomial $V(X) = 1 + 1X + 1X^2 + 1X^3 + 1X^4 + 1X^5 + 1X^6 + 1X^7$							
Decoded		**Encoded**		**Linear combination of encodings**	**Scaled Encoded Combination**	**Bootstrapping**	**Decoded result**
d_1	d_2	m_1	m_2	$m_1 + m_2 - 1$	$\hat{\Delta} \cdot (m_1 + m_2 - 1)$	$V(X) \cdot X^{-\hat{\Delta} \cdot (m_1 + m_2 - 1) + e}$	$d_1 \wedge d_2$
0	0	-1	-1	-3	-6	constant term's coeff. is -1	0
0	1	-1	1	-1	-2	constant term's coeff. is -1	0
1	0	1	-1	-1	-2	constant term's coeff. is -1	0
1	1	1	1	1	2	constant term's coeff. is 1	1

Table 5: An example truth table for an AND operation with an additional encoding.

?? is a programmable bootstrapping design for an AND logic gate. For this application, we define the LUT polynomial V as $V(X) = \sum_{i=0}^{7} -X^i$. The LUT polynomial $V(X)$ maps one half of the plaintext domain to -1, while the other half to -1 (as the terms wrap around the boundary of $n = 7$). In this design setup, each bit is separately encrypted as independent TFHE ciphertext. Gate inputs 0 and 1 are encoded as -1 and 1, respectively. The linear combination (i.e., homomorphic computation formula) for an AND gate is $\mathsf{LWE}_{\vec{s},\sigma}(\Delta m_1) + \mathsf{LWE}_{\vec{s},\sigma}(\Delta m_2) - 1$. Its output is positive if both inputs are positive (i.e. 1, in which case the blind rotation will rotate V to the left by $\hat{\Delta} \cdot 1 + e$ positions and the constant term's coefficient will be 1. Thus, the output of blind rotation and coefficient extraction will be $\mathsf{LWE}_{\vec{s},\sigma}(\Delta \cdot 1)$ with a reduced noise, which is an encoding of 1. This design can tolerate the maximum noise of $|e| = 1$. To endure bigger noises, we should increase q and n.

Note that the AND gate's LUT layout is negacyclic, which is a special case, thus we could use the entire $2n = 16$ coefficient states in $V(X)$ for the AND gate mapping function's outputs, by leveraging $V(X)$'s innate property of negacyclic rotation. However, in many use cases, the LUT layout is not necessarily negacyclic like this AND gate example. Even our noise bootstrapping's LUT layout (**??**) was not negacyclic, but a unity function (as it simply removes the noise). Thus, for most use cases, we need to use only $\frac{t}{2}$ out of t plaintext space to avoid more than $n - 1$ rotations of $V(X)$ (**??**).

Besides the AND gate, other logic gates can be built in a similar manner, each of which is based on a different linear combination formula and LUT layout.

Division: TFHE does not support direct division of plaintext numbers of any size. This is because

TFHE's LWE vector elements are in the Z_q ring, where each element g does not necessarily have a multiplicative inverse g^{-1}, which makes it hard to multiply g^{-1} to the target number to divide. Instead, division can be implemented as binary division based on the gates implemented by gate bootstrapping. To support binary division, each plaintext has to be a single bit and encrypted as an independent ciphertext. Or multiple bits can be bundled up and processed concurrently by designing a linear combination formula, similar to the linear combination that we designed for processing 2 input bits of an AND gate.

D-1.8.10 Application: Neural Networks Bootstrapping

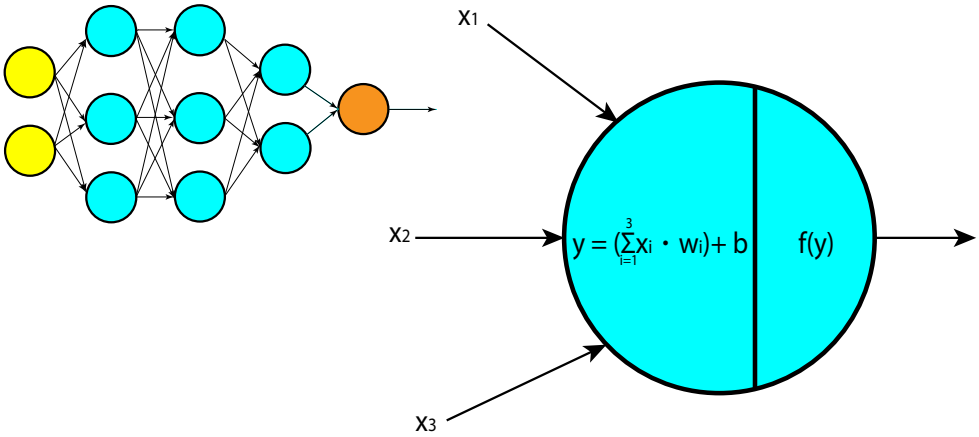


Figure 14: An illustration of neural networks

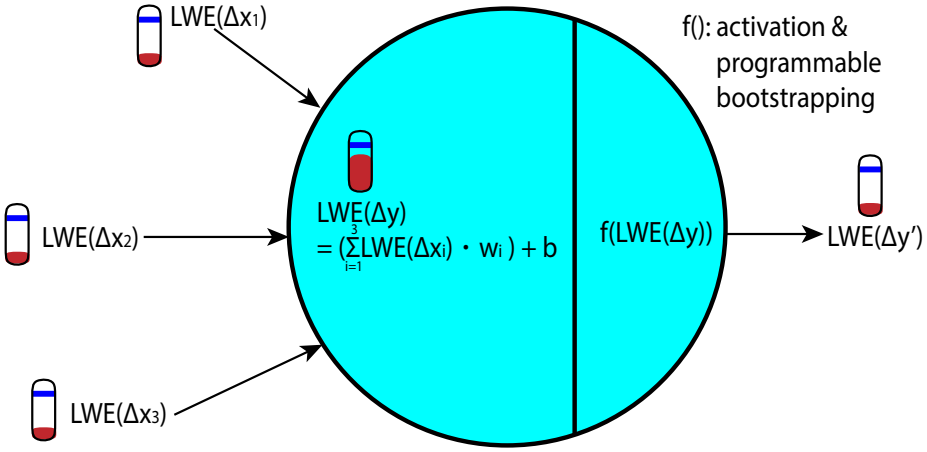


Figure 15: An illustration of neural networks's programmable bootstrapping

Homomorphic encryption can be applied to the neurons of deep neural networks, in which each neuron is generally comprised of two steps of computation:

1. **Linear Combination of Input Values:** An input feature value (or intermediate value) set $(x_1, x_2, \cdots, x_n)$, a weight set $(w_1, w_2, \cdots x, w_n)$, and a bias b are computed as: $y = \sum_{i=1}^{n} a_0 w_0 + b$.
2. **Activation Function:** $f(y)$ is computed, where f is a non-linear activation function such as the sin function, ReLU, sigmoid, hyperbolic tangent, etc.

TFHE can homomorphically compute the 1st step's linear combination formula: $y = \sum_{i=1}^{n} a_0 w_0 + b$ as $\sum_{i=1}^{n} \mathsf{LWE}_{\vec{s},\sigma}(a_0) \cdot w_0 + b$, which can be implemented as ciphertext addition (**??**) and ciphertext-to-plaintext multiplication (**??**).

However, the 2nd step's non-linear functions cannot be expressed as addition and multiplication of ciphertexts. To address this issue, the activation function can be evaluated as a programmable bootstrapping, such that the output of the bootstrapping matches or (or is similar) to the output of the activation function. If we use bootstrapping at the 2nd step, noises can be refreshed at the end of every neuron, thus we can potentially handle neural networks of any depth without worrying about the noise growth.

D-1.9 TFHE on a Discrete Torus

- Reference: Guide to Fully Homomorphic Encryption over the [Discretized] Torus [**?**]

Torus $\mathbb{T}$ is a continuous real number domain between 0 and 1 that wraps around, that is $[0, 1)$.
A discrete torus $\mathbb{T}_t$ is a finite real number set: $\left(0, \frac{1}{t}, \frac{2}{t}, \cdots, \frac{t-1}{t}\right)$
In the previous subsections, we explained the TFHE scheme based on the following setup:

$m \in \mathbb{Z}_t \ \vec{s} = \{0,1\}^k \ e \in \mathbb{Z}_q$
$\mathsf{LWE}_{\vec{s},\sigma}(\Delta m) = (a_0, a_1, \cdots, a_k, b) \in \mathbb{Z}_t^{k+1}$

However, the original TFHE scheme is designed based on a discrete torus:

$m \in \mathbb{T}_t, \ \vec{s} \in \{0,1\}^k, \ e \in \mathbb{T}_q$

$\mathsf{LWE}_{\vec{s},\sigma}(m) = \mathsf{ct} = (a_0, a_1, \cdots, a_k, b) \in \mathbb{T}_q^{k+1}$
$b = \sum_{i=0}^{k} (a_i s_i) + m + e \in \mathbb{T}_q$
$\mathsf{LWE}^{-1}_{\vec{s},\sigma}(\mathsf{ct}) = \left\lceil b - \sum_{i=0}^{k-1} (a_i s_i) \right\rfloor_{\frac{1}{t}} = \left\lceil m + e \right\rfloor_{\frac{1}{t}} = m$, given $e < \frac{1}{2t}$
where $\lceil x \rfloor_{\frac{1}{t}}$ means rounding x to the nearest multiple of $\frac{1}{t}$

The original TFHE's difference is that all values (either polynomial coefficients or vector elements) are computed in a floating point modulo 1 (i.e., $[0, 1)$) instead of a big integer (i.e., $[0, q)$). This means the plaintext also has to be encoded as values within $[0, 1)$ instead of integers within $[0, q)$. Note that in the original TFHE scheme, there is no need for the scaling factor Δ, because the continuous domain of torus $[0, 1)$ provides a floating-point precision up to q discrete decimal values, and its decryption process can successfully blow away the noise E as far as each coefficient (or vector element) e_i in E is smaller than $\frac{1}{2t}$.

Both the torus-based and integer-ring-based TFHE schemes are built based on the same fundamental principles.

D-2 BFV Scheme

The BFV scheme is designed for homomorphic addition and multiplication of integers. BFV's encoding scheme does not require such approximation issues because BFV is designed to encode only integers. Therefore, BFV guarantees exact encryption and decryption. BFV is suitable for use cases where the encrypted and decrypted values should exactly match (e.g., voting, financial computation), whereas CKKS is suitable for the use cases that tolerate tiny errors (e.g., data analytics, machine learning).

In BFV, each plaintext is encrypted as an RLWE ciphertext. Therefore, BFV's ciphertext-to-ciphertext addition, ciphertext-to-plaintext addition, and ciphertext-to-plaintext multiplication are implemented based on GLWE's homomorphic addition and multiplication (as we learned in **??**), with $k = 1$ to make GLWE an RLWE.

Required Background

- ??: ??
- ??: ??
- ??: ??
- ??: ??
- ??: ??
- ??: ??
- ??: ??
- ??: ??
- ??: ??
- ??: ??
- ??: ??
- ??: ??
- ??: ??
- ??: ??
- ??: ??
- ??: ??
- ??: ??
- ??: ??
- ??: ??
- ??: ??
- ??: ??
- ??: ??
- ??: ??

D-2.1 Single Value Encoding

BFV supports two encoding schemes: single value encoding and batch encoding. In this subsection, we will explain the single value encoding scheme.

⟨Summary ??⟩ BFV Encoding

Input Integer: Decompose the input integer m as follows:
$m = b_{n-1} \cdot 2^{n-1} + b_{n-2} \cdot 2^{n-2} + \cdots + b_1 \cdot 2^1 + b_0 \cdot 2^0$, where each $b_i \in \mathbb{Z}_q$

(Note that there is more than 1 way to decompose the same m)

Encoded Polynomial: $M(X) = b_0 + b_1 X + b_2 X^2 + \cdots + b_{n-1} X^{n-1} \in \mathcal{R}_{\langle n,q \rangle}$

Decoding: $M(X = 2) = m$

Let's analyze whether the encoding scheme in Summary **??** preserves isomorphism: homomorphic and bijective.

Homomorphism: Let's denote the above encoding scheme as the mapping σ. Suppose we have two integers m_1, m_2 as follows:

$m_1 = \sum_{i=0}^{n-1} (b_{1,i} \cdot 2^i), \quad m_2 = \sum_{i=0}^{n-1} (b_{2,i} \cdot 2^i)$

Then,

$$\sigma(m_1 + m_2) = \sigma\left(\sum_{i=0}^{n-1} (b_{1,i} \cdot 2^i) + \sum_{i=0}^{n-1} (b_{2,i} \cdot 2^i)\right)$$
$$= \sigma\left(\sum_{i=0}^{n-1} (b_{1,i} + b_{2,i}) \cdot 2^i\right)$$
$$= \sum_{i=0}^{n-1} (b_{1,i} + b_{2,i}) \cdot X^i$$
$$= \sum_{i=0}^{n-1} b_{1,i} \cdot X^i + \sum_{i=0}^{n-1} b_{2,i} \cdot X^i$$
$$= \sigma\left(\sum_{i=0}^{n-1} (b_{1,i} \cdot 2^i)\right) + \sigma\left(\sum_{i=0}^{n-1} (b_{2,i} \cdot 2^i)\right)$$
$$= \sigma(m_1) + \sigma(m_2)$$

$$\sigma(m_1 \cdot m_2) = \sigma\left(\sum_{i=0}^{n-1} (b_{1,i} \cdot 2^i) \cdot \sum_{i=0}^{n-1} (b_{2,i} \cdot 2^i)\right)$$
$$= \sigma\left(\sum_{i=0}^{2\cdot(n-1)} \sum_{j=0}^{i} (b_{1,j} \cdot b_{2,i-j}) \cdot 2^i\right)$$
$$= \sum_{i=0}^{2\cdot(n-1)} \sum_{j=0}^{i} (b_{1,j} \cdot b_{2,i-j}) \cdot X^i$$
$$= \sum_{i=0}^{n-1} b_{1,i} \cdot X^i \cdot \sum_{i=0}^{n-1} b_{2,i} \cdot X^i$$
$$= \sigma\left(\sum_{i=0}^{n-1} (b_{1,i} \cdot 2^i)\right) \cdot \sigma\left(\sum_{i=0}^{n-1} (b_{2,i} \cdot 2^i)\right)$$

$= \sigma(m_1) \cdot \sigma(m_2)$

Since $\sigma(m_1 + m_2) = \sigma(m_1) + \sigma(m_2)$ and $\sigma(m_1 \cdot m_2) = \sigma(m_1) \cdot \sigma(m_2)$, the encoding scheme in Summary **??** is homomorphic.

One-to-Many Mappings: This encoding scheme preserves one-to-many mappings between the input integers and the encoded polynomials, because there is more than 1 way to encode the same input integer m, which can be encoded as different polynomials. For example, suppose that we have the following two input integers and their summation: $m_1 = 0111$, $m_2 = 0010$, $m_{1+2} = m_1 + m_2 = 1001$. Then, their encoded polynomials are as follows:
$M_1(X) = 0X^3 + 1X^2 + 1X + 1$
$M_2(X) = 0X^3 + 0X^2 + 1X + 0$
$M_{1+2}(X) = 0X^3 + 1X^2 + 2X + 1$
However, the following polynomial is also mapped to $m_{1+2} = 1001$:
$M_3(X) = 1X^3 + 0X^2 + 0X^1 + 1$

That being said, both polynomials are mapped to the same m_{1+2} (i.e., 1001) as follows:
$M_{1+2}(2) = 4 + 4 + 1 = 9$
$M_3(2) = 8 + 1 = 9$

Although the encoding scheme in Summary **??** is not bijective but only one-to-many mappings, it preserves homomorphism and the decoded number decomposition sums to the same original integer. Therefore, this encoding scheme can be validly used for fully homomorphic encryption.

D-2.2 Batch Encoding

While the single-value encoding scheme (**??**) encodes & decodes each individual value one at a time, the batch encoding scheme does the same for a huge list of values simultaneously using a large dimensional vector. Therefore, batch encoding is more efficient than single-value encoding. Furthermore, batch-encoded values can be homomorphically added or multiplied simultaneously element-wise by vector-to-vector addition and Hadamard product. Therefore, the homomorphic operation of batch-encoded values can be processed more efficiently in a SIMD (single-instruction-multiple-data) manner than single-value encoded ones.

BFV's encoding converts an n-dimensional integer input slot vector $\vec{v} = (v_0, v_1, v_2, \cdots v_{n-1})$ modulo t into another n-dimensional vector $\vec{m} = (m_0, m_1, m_2, \cdots m_{n-1})$ modulo t, which are the coefficients of the encoded $(n-1)$-degree (or lesser-degree) polynomial $M(X) \in \mathbb{Z}_t[X]/(X^n + 1)$.

D-2.2.1 Encoding$_1$

In **??**, we learned that an $(n-1)$-degree (or lesser degree) polynomial can be isomorphically mapped to an n-dimensional vector based on the mapping σ (we notate σ_c in **??** as σ for simplicity):
$\sigma : M(X) \in \mathbb{Z}_t[X]/(X^n + 1) \longrightarrow (M(\omega), M(\omega^3), M(\omega^5), \cdots, M(\omega^{2n-1})) \in \mathbb{Z}_t^n$

, which evaluates the polynomial $M(X)$ at n distinct ($\mu = 2n$)-th primitive roots of unity: $\omega, \omega^3, \omega^5, \cdots, \omega^{2n-1}$. Let $\vec{m}$ be a vector that contains n coefficients of the polynomial $M(X)$. Then, we can express the mapping σ as follows:

$\vec{v} = W^T \cdot \vec{m}$

, where W^T is as follows:

$$W^T = \begin{bmatrix} 1 & (\omega) & (\omega)^2 & \cdots & (\omega)^{n-1} \\ 1 & (\omega^3) & (\omega^3)^2 & \cdots & (\omega^3)^{n-1} \\ 1 & (\omega^5) & (\omega^5)^2 & \cdots & (\omega^5)^{n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & (\omega^{2n-1}) & (\omega^{2n-1})^2 & \cdots & (\omega^{2n-1})^{n-1} \end{bmatrix}$$

W^T is a transpose of W described in **??**

Note that the dot product between each row of W^T and $\vec{m}$ computes the evaluation of $M(X)$ at each $X = \{w, w^3, w^5, \cdots, w^{2n-1}\}$. In the BFV encoding scheme, the $\mathsf{Encoding}_1$ process encodes an n-dimensional input slot vector $\vec{v} \in \mathbb{Z}_t$ into a plaintext polynomial $M(X) \in \mathbb{Z}_t[X]/(X^n + 1)$, and the $\mathsf{Decoding}_2$ process decodes $M(X)$ back to $\vec{v}$. Since $W^T \cdot \vec{m}$ gives us $\vec{v}$ which is a decoding of $M(X)$, we call W^T a decoding matrix. Meanwhile, the goal of $\mathsf{Encoding}_1$ is to encode $\vec{v}$ into $M(X)$ so that we can do homomorphic computations based on $M(X)$. Given the relation $\vec{v} = W^T \cdot \vec{m}$, the encoding formula can be derived as follows:

$(W^T)^{-1} \cdot \vec{v} = (W^T)^{-1} W^T \cdot \vec{m}$

$\vec{m} = (W^T)^{-1} \cdot \vec{v}$

Therefore, we need to find out what $(W^T)^{-1}$ is, the inverse of W^T as the encoding matrix. But we already learned from Theorem **??** (in **??**) that $V^{-1} = \frac{V^T \cdot I_n^R}{n}$, where $V = W^T$ and $V = W$. In other words, $(W^T)^{-1} = \frac{W \cdot I_n^R}{n}$. Therefore, we can express the BFV encoding formula as:

$$\vec{m} = (W^T)^{-1} \cdot \vec{v} = \frac{W \cdot I_n^R \cdot \vec{v}}{n}, \quad \text{where}$$

$$W = \begin{bmatrix} 1 & 1 & 1 & \cdots & 1 \\ (\omega) & (\omega^3) & (\omega^5) & \cdots & (\omega^{2n-1}) \\ (\omega)^2 & (\omega^3)^2 & (\omega^5)^2 & \cdots & (\omega^{2n-1})^2 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ (\omega)^{n-1} & (\omega^3)^{n-1} & (\omega^5)^{n-1} & \cdots & (\omega^{2n-1})^{n-1} \end{bmatrix}$$

$$= \begin{bmatrix} 1 & 1 & \cdots & 1 & 1 & \cdots & 1 & 1 \\ (\omega) & (\omega^3) & \cdots & (\omega^{\frac{n}{2}-1}) & (\omega^{-(\frac{n}{2}-1)}) & \cdots & (\omega^{-3}) & (\omega^{-1}) \\ (\omega)^2 & (\omega^3)^2 & \cdots & (\omega^{\frac{n}{2}-1})^2 & (\omega^{-(\frac{n}{2}-1)})^2 & \cdots & (\omega^{-3})^2 & (\omega^{-1})^2 \\ \vdots & \vdots & \vdots & \ddots & \vdots & & & \\ (\omega)^{n-1} & (\omega^3)^{n-1} & \cdots & (\omega^{\frac{n}{2}-1})^{n-1} & (\omega^{-(\frac{n}{2}-1)})^{n-1} & \cdots & (\omega^{-3})^{n-1} & (\omega^{-1})^{n-1} \end{bmatrix}$$

, where $\omega = g^{\frac{t-1}{2n}} \bmod t$ (g is a generator of $\mathbb{Z}_t^\times$). In **??**, we learned that W is a valid basis of the n-dimensional vector space. Therefore, $\frac{W \cdot \vec{v}}{n} = \vec{m}$ is guaranteed to be a unique vector corresponding to each $\vec{v}$ in the n-dimensional vector space $\mathbb{Z}_t^n$ (refer to Theorem **??** in **??**), and thereby the polynomial $M(X)$ comprising the n elements of $\vec{m}$ as coefficients is a unique polynomial bi-jective

to $\vec{v}$.

Note that by computing $\frac{W \cdot I_n^R \cdot \vec{v}}{n}$, we transform the input slot vector $\vec{v}$ into another vector $\vec{m}$ in the same vector space $\mathbb{Z}_t^n$, while preserving isomorphism between these two vectors (i.e., bi-jective one-to-one mappings and homomorphism on the $(+, \cdot)$ operations).

D-2.2.2 Encoding$_2$

Once we have the n-dimensional vector $\vec{m}$, we scale (i.e., multiply) it by some scaling factor $\Delta = \left\lfloor \frac{q}{t} \right\rfloor$, where q is the ciphertext modulus. We scale $\vec{m}$ by Δ and make it $\Delta\vec{m}$. The n integers in $\Delta\vec{m}$ will be used as n coefficients of the plaintext polynomial for RLWE encryption. The finally encoded plaintext polynomial $\Delta M = \sum_{i=0}^{n-1} \Delta m_i X^i$.

D-2.2.3 Decoding$_1$

Once an RLWE ciphertext is decrypted to $\Delta M = \sum_{i=0}^{n-1} \Delta m_i X^i$, we compute $\frac{\Delta\vec{m}}{\Delta} = \vec{m}$.

D-2.2.4 Decoding$_2$

In **??**, we already derived the decoding formula that transforms an $(n-1)$-degree polynomial having integer modulo t coefficients into an n-dimensional input slot vector as follows:
$\vec{v} = W^T \cdot \vec{m}$

D-2.2.5 Summary

⟨Summary ??⟩ BFV's Encoding and Decoding

Input: An n-dimensional integer modulo t vector $\vec{v} = (v_0, v_1, \cdots, v_{n-1}) \in \mathbb{Z}_t^n$

Encoding

1. Convert $\vec{v} \in \mathbb{Z}_t^n$ into $\vec{m} \in \mathbb{Z}_t^n$ by applying the transformation $\vec{m} = n^{-1} \cdot W \cdot I_n^R \cdot \vec{v}$, where W is a basis of the n-dimensional vector space crafted as follows:

$$W = \begin{bmatrix} 1 & 1 & 1 & \cdots & 1 \\ (\omega) & (\omega^3) & (\omega^5) & \cdots & (\omega^{2n-1}) \\ (\omega)^2 & (\omega^3)^2 & (\omega^5)^2 & \cdots & (\omega^{2n-1})^2 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ (\omega)^{n-1} & (\omega^3)^{n-1} & (\omega^5)^{n-1} & \cdots & (\omega^{2n-1})^{n-1} \end{bmatrix}$$

$$= \begin{bmatrix} 1 & 1 & \cdots & 1 & 1 & \cdots & 1 & 1 \\ (\omega) & (\omega^3) & \cdots & (\omega^{\frac{n}{2}-1}) & (\omega^{-(\frac{n}{2}-1)}) & \cdots & (\omega^{-3}) & (\omega^{-1}) \\ (\omega)^2 & (\omega^3)^2 & \cdots & (\omega^{\frac{n}{2}-1})^2 & (\omega^{-(\frac{n}{2}-1)})^2 & \cdots & (\omega^{-3})^2 & (\omega^{-1})^2 \\ \vdots & \vdots & \vdots & \ddots & \vdots & & & \\ (\omega)^{n-1} & (\omega^3)^{n-1} & \cdots & (\omega^{\frac{n}{2}-1})^{n-1} & (\omega^{-(\frac{n}{2}-1)})^{n-1} & \cdots & (\omega^{-3})^{n-1} & (\omega^{-1})^{n-1} \end{bmatrix}$$

, where $\omega = g^{\frac{t-1}{2n}} \bmod t$ (g is a generator of $\mathbb{Z}_t^\times$)

2. Convert $\vec{m}$ into a scaled integer vector $\Delta\vec{m}$, where $1 \leq \Delta \leq \lfloor \frac{q}{t} \rfloor$ is a scaling factor. If Δ is too close to 1, the noise budget will become too small. If Δ is too close to $\lfloor \frac{q}{t} \rfloor$, the plaintext budget will become too small (i.e., cannot wrap around t much). The finally encoded plaintext polynomial $\Delta M = \sum_{i=0}^{n-1} \Delta m_i X^i \in \mathbb{Z}_q[X]/(X^n+1)$.

Decoding: From the plaintext polynomial $\Delta M = \sum_{i=0}^{n-1} \Delta m_i X^i$, recover $\vec{m} = \frac{\Delta\vec{m}}{\Delta}$. Then, compute $\vec{v} = W^T \cdot \vec{m}$.

However, Summary **??** is not the final version of BFV's batch encoding. In **??**, we will explain how to homomorphically rotate the input vector slots without decrypting the ciphertext that encapsulates it. To support such homomorphic rotation, we will need to slightly update the encoding scheme explained in Summary **??**. We will explain how to do this in **??**, and BFV's final encoding scheme is summarized in Summary **??** in **??**.

D-2.3 Encryption and Decryption

BFV encrypts and decrypts ciphertexts based on the RLWE cryptosystem (**??**) with the sign of each $A \cdot S$ term flipped in the encryption and decryption formula. Specifically, this is equivalent to the alternative version of the GLWE cryptosystem (**??**) with $k = 1$. Thus, BFV's encryption and decryption formulas are as follows:

⟨Summary ??⟩ BFV Encryption and Decryption

Initial Setup: $\Delta = \left\lfloor \frac{q}{t} \right\rfloor$ is a plaintext scaling factor for polynomial encoding, $S \xleftarrow{\$} \mathcal{R}_{\langle n,2 \rangle}$, where plaintext modulus t is either a prime (p) or a power of prime (p^r), and ciphertext modulus $q \gg t$. As for the coefficients of polynomial S, they can be either binary (i.e., $\{0,1\}$) or ternary (i.e., $\{-1,0,1\}$).

Encryption Input: $\Delta M \in \mathcal{R}_{\langle n,q \rangle}, A_i \xleftarrow{\$} \mathcal{R}_{\langle n,q \rangle}, E \xleftarrow{\chi_\sigma} \mathcal{R}_{\langle n,q \rangle}$

1. Compute $B = -A \cdot S + \Delta M + E \in \mathcal{R}_{\langle n,q \rangle}$
2. $\mathsf{RLWE}_{S,\sigma}(\Delta M) = (A, B) \in \mathcal{R}^2_{\langle n,q \rangle}$

Decryption Input: $\mathsf{ct} = (A, B) \in \mathcal{R}^2_{\langle n,q \rangle}$

$$\mathsf{RLWE}^{-1}_{S,\sigma}(\mathsf{ct}) = \left\lceil \frac{B + A \cdot S \bmod q}{\Delta} \right\rfloor \bmod t = \left\lceil \frac{\Delta M + E}{\Delta} \right\rfloor \bmod t = M \bmod t$$

(The noise $E = \sum_{i=0}^{n-1} e_i X^i$ gets eliminated by the rounding process)

Conditions for Correct Decryption:

1. Each noise coefficient e_i growing during homomorphic operations should never exceed:

$e_i < \frac{\Delta}{2}$.

2. Each plaintext coefficient m_i being updated across homomorphic operations should never overflow or underflow q (i.e., $m_i < q$)
3. The specific noise bound is as explained in **??**:
   $\frac{k_i t + e_i}{\lfloor \frac{q}{t} \rfloor} < \frac{1}{2}$ # $k_i t$ accounts for the plaintext m_i's wrapped-around value as multiples of t

D-2.4 Ciphertext-to-Ciphertext Addition

BFV's ciphertext-to-ciphertext addition uses RLWE's ciphertext-to-ciphertext addition scheme with the sign of the $A \cdot S$ term flipped in the encryption and decryption formula. Specifically, this is equivalent to the alternative GLWE version's (**??**) ciphertext-to-ciphertext addition scheme with $k = 1$.

⟨Summary ??⟩ BFV Ciphertext-to-Ciphertext Addition

$$\begin{aligned}
&\mathsf{RLWE}_{S,\sigma}(\Delta M^{\langle 1 \rangle}) + \mathsf{RLWE}_{S,\sigma}(\Delta M^{\langle 2 \rangle}) \\
&= (A^{\langle 1 \rangle},\ B^{\langle 1 \rangle}) + (A^{\langle 2 \rangle},\ B^{\langle 2 \rangle}) \\
&= (A^{\langle 1 \rangle} + A^{\langle 2 \rangle},\ B^{\langle 1 \rangle} + B^{\langle 2 \rangle}) \\
&= \mathsf{RLWE}_{S,\sigma}(\Delta(M^{\langle 1 \rangle} + M^{\langle 2 \rangle}))
\end{aligned}$$

D-2.5 Ciphertext-to-Plaintext Addition

BFV's ciphertext-to-plaintext addition uses RLWE's ciphertext-to-plaintext addition scheme with the sign of the $A \cdot S$ term flipped in the encryption and decryption formula. Specifically, this is equivalent to the alternative GLWE version's (**??**) ciphertext-to-plaintext addition scheme (**??**) with $k = 1$.

⟨Summary ??⟩ BFV Ciphertext-to-Plaintext Addition

$$\begin{aligned}
&\mathsf{RLWE}_{S,\sigma}(\Delta M) + \Delta\Lambda \\
&= (A,\ B) + \Delta\Lambda \\
&= (A,\ B + \Delta \cdot \Lambda) \\
&= \mathsf{RLWE}_{S,\sigma}(\Delta(M + \Lambda))
\end{aligned}$$

D-2.6 Ciphertext-to-Plaintext Multiplication

BFV's ciphertext-to-plaintext multiplication uses RLWE's ciphertext-to-plaintext multiplication scheme with the sign of the $A \cdot S$ term flipped in the encryption and decryption formula. Specifically, this is equivalent to the alternative GLWE version's (**??**) ciphertext-to-plaintext multiplication scheme (**??**) with $k = 1$.

⟨Summary ??⟩ BFV Ciphertext-to-Plaintext Multiplication

$\mathsf{RLWE}_{S,\sigma}(\Delta M) \cdot \Lambda$
$= (A,\ B) \cdot \Lambda$
$= (A \cdot \Lambda,\ B \cdot \Lambda)$
$= \mathsf{RLWE}_{S,\sigma}(\Delta(M \cdot \Lambda))$

D-2.7 Ciphertext-to-Ciphertext Multiplication

- **Reference 1:** Introduction to the BFV encryption scheme [?]
- **Reference 2:** Somewhat Partially Fully Homomorphic Encryption [?]

Given two ciphertexts $\mathsf{RLWE}_{S,\sigma}(\Delta M^{\langle 1\rangle})$ and $\mathsf{RLWE}_{S,\sigma}(\Delta M^{\langle 2\rangle})$, the goal of ciphertext-to-ciphertext multiplication is to derive a new ciphertext whose decryption is $\Delta M^{\langle 1\rangle} M^{\langle 2\rangle}$. Ciphertext-to-ciphertext multiplication is more complex than ciphertext-to-plaintext multiplication. It comprises four steps: (1) ModRaise; (2) polynomial multiplication; (3) relinearization; and (4) rescaling.

For better understanding, we will explain BFV's ciphertext-to-ciphertext multiplication based on the alternate version of RLWE (Theorem **??** in **??**), where the sign of the AS term is flipped in the encryption and decryption formulas.

D-2.7.1 ModRaise

We learned from Summary **??** (in **??**) that a BFV ciphertext whose ciphertext modulus is q has the (decryption) relation: $\Delta M + E = A \cdot S + B - K \cdot q$, where $K \cdot q$ stands for modulo reduction by q. ModRaise is a process of forcibly modifying the modulus of a ciphertext from $q \to Q$, where $q \ll Q$. Suppose we modify the modulus of ciphertext (A, B) from q to Q, where $Q = q \cdot \Delta$ (remember $\Delta = \left\lfloor \frac{q}{t} \right\rfloor$). Then, the decryption of the *mod-raised* ciphertext will output $A \cdot S + B \bmod Q$. However, since each polynomial coefficient of A and B is less than q and each polynomial coefficient of S is either $\{-1, 0, 1\}$, the resulting polynomial of $A \cdot S + B$ is guaranteed to have each coefficient strictly less than Q even without modulo reduction by Q– this is because $(q-1) \cdot n + (q-1) < Q$, where $(q-1) \cdot n$ is the maximum possible coefficient of $A \cdot S$ and $(q-1)$ is the maximum possible coefficient of B. And as mentioned before, we know the relation: $A \cdot S + B = \Delta M + E + Kq$. Therefore, the decryption of the *mod-raised* ciphertext (A, B) mod Q is as follows:

$\Delta M + E + Kq \bmod Q = \Delta M + E + Kq$ # since $\Delta M + E + Kq < Q$

The first step of BFV's ciphertext-to-ciphertext multiplication is to *mod-raise* the two input ciphertexts $(A^{\langle 1\rangle}, B^{\langle 1\rangle})$ mod q and $(A^{\langle 2\rangle}, B^{\langle 2\rangle})$ mod q from $q \to Q$ (where $Q = q \cdot \Delta$) as follows:

$(A^{\langle 1\rangle}, B^{\langle 1\rangle}) \bmod Q$
$(A^{\langle 2\rangle}, B^{\langle 2\rangle}) \bmod Q$

After ModRaise, the decryption of these two ciphertexts would be the following:

$A^{\langle 1\rangle} \cdot S + B^{\langle 1\rangle} = \Delta M^{\langle 1\rangle} + E^{\langle 1\rangle} + K_1 q < Q$
$A^{\langle 2\rangle} \cdot S + B^{\langle 2\rangle} = \Delta M^{\langle 2\rangle} + E^{\langle 2\rangle} + K_2 q < Q$

Therefore, the *mod-raised* ciphertexts have the following form:

$\mathsf{RLWE}_{S,\sigma}(\Delta M^{\langle 1\rangle} + K_1 q) = (A^{\langle 1\rangle}, B^{\langle 1\rangle}) \bmod Q$

$\mathsf{RLWE}_{S,\sigma}(\Delta M^{\langle 2\rangle} + K_2 q) = (A^{\langle 2\rangle}, B^{\langle 2\rangle}) \bmod Q$

D-2.7.2 Polynomial Multiplication

Our next goal is to derive a new ciphertext which encrypts $(\Delta M^{\langle 1\rangle} + E^{\langle 1\rangle} + K_1 q) \cdot (\Delta M^{\langle 2\rangle} + E^{\langle 2\rangle} + K_2 q)$.

First, we can derive the following relation:

$$\begin{aligned}
&(\Delta M^{\langle 1\rangle} + E^{\langle 1\rangle} + K_1 q) \cdot (\Delta M^{\langle 2\rangle} + E^{\langle 2\rangle} + K_2 q) \\
&= (A^{\langle 1\rangle} \cdot S + B^{\langle 1\rangle}) \cdot (A^{\langle 2\rangle} \cdot S + B^{\langle 2\rangle}) \\
&= \underbrace{B^{\langle 1\rangle} B^{\langle 2\rangle}}_{D_0} + \underbrace{(B^{\langle 2\rangle} A^{\langle 1\rangle} + B^{\langle 1\rangle} A^{\langle 2\rangle})}_{D_1} \cdot S + \underbrace{(A^{\langle 1\rangle} \cdot A^{\langle 2\rangle})}_{D_2} \cdot S \cdot S \\
&= D_0 + D_1 \cdot S + D_2 \cdot S^2
\end{aligned}$$

Meanwhile, we also have the following relations:

$\mathsf{RLWE}^{-1}_{S,\sigma}(\Delta M^{\langle 1\rangle} + K_1 q) = \Delta M^{\langle 1\rangle} + E^{\langle 1\rangle} + K_1 q$

$\mathsf{RLWE}^{-1}_{S,\sigma}(\Delta M^{\langle 2\rangle} + K_2 q) = \Delta M^{\langle 2\rangle} + E^{\langle 2\rangle} + K_2 q$

Combining all these, we reach the following relation:

$\mathsf{RLWE}^{-1}_{S,\sigma}(\Delta M^{\langle 1\rangle} + K_1 q) \cdot \mathsf{RLWE}^{-1}_{S,\sigma}(\Delta M^{\langle 2\rangle} + K_2 q) = D_0 + D_1 \cdot S + D_2 \cdot S^2$

Notice that D_0, D_1, and D_2 are known values as ciphertext components, whereas S is only known to the private key owner. Therefore, we can view $D_0 + D_1 \cdot S + D_2 \cdot S^2$ as a decryption formula such that given the ciphertext components D_0, D_1, D_2 and the private key S, one can derive $(\Delta M^{\langle 1\rangle} + E^{\langle 1\rangle} + K_1 q) \cdot (\Delta M^{\langle 2\rangle} + E^{\langle 2\rangle} + K_2 q)$. In other words, we can let (D_0, D_1, D_2) be a new form of ciphertext which can be decrypted by S into $(\Delta M^{\langle 1\rangle} + E^{\langle 1\rangle} + K_1 q) \cdot (\Delta M^{\langle 2\rangle} + E^{\langle 2\rangle} + K_2 q)$.

However, (D_0, D_1, D_2) is not in the RLWE ciphertext format, because it has 3 components instead of 2. Having 3 ciphertext components is computationally inefficient, as its decryption involves a square root of S (i.e., S^2). Over consequent ciphertext-to-ciphertext multiplications, this S term will double its exponents as $S^4, S^8, \cdots$ as well as the number of ciphertext components, which would exponentially increase the computational overhead of decryption. Therefore, we want to convert the intermediate ciphertext format (D_0, D_1, D_2) into a regular BFV ciphertext format that has two polynomials as ciphertext components. This conversion process is called a relinearization process (which will be explained in the next subsection).

D-2.7.3 Relinearization

Relinearization is a process of converting the polynomial triplet $(D_0, D_1, D_2) \in \mathcal{R}^3_{\langle n,Q\rangle}$ into two RLWE ciphertexts ct_α and ct_β which hold the relation: $D_0 + D_1 S + D_2 S^2 = \mathsf{RLWE}^{-1}_{S,\sigma}(\mathsf{ct}_\alpha + \mathsf{ct}_\beta)$.

In the formula $D_0 + D_1 S + D_2 S^2$, we can re-write $D_0 + D_1 S$ as a *synthetic* RLWE ciphertext $\mathsf{ct}_\alpha = (D_1, D_0)$, which can be decrypted by S into $D_1 S + D_0$. Similarly, our next task is to derive a synthetic RLWE ciphertext ct_β whose decryption is $D_2 \cdot S^2$ (i.e., $\mathsf{RLWE}^{-1}_{S,\sigma}(\mathsf{ct}_\beta) = D_2 \cdot S^2$).

A naive way of creating a ciphertext that encrypts $D_2 \cdot S^2$ is as follows: we encrypt S^2 into an RLWE ciphertext as $\mathsf{RLWE}_{S,\sigma}(S^2) = (A^{\langle s\rangle}, B^{\langle s\rangle})$ such that $A^{\langle s\rangle} \cdot S + B^{\langle s\rangle} = S^2 + E^{\langle s\rangle} \bmod Q$ (where the ciphertext modulus is Q and the plaintext scaling factor $\Delta = 1$). Then, we perform a ciphertext-to-plaintext multiplication (**??**) with D_2, treating D_2 as a plaintext polynomial in

modulo Q. However, this approach does not work in practice, because computing $D_2 \cdot \mathsf{RLWE}_{S,\sigma}(S^2)$ generates a huge noise as follows:

$D_2 \cdot (A^{\langle s \rangle}, B^{\langle s \rangle}) = (D_2 \cdot A^{\langle s \rangle}, D_2 \cdot B^{\langle s \rangle})$

, whose decryption is:

$D_2 \cdot A^{\langle s \rangle} \cdot S + D_2 \cdot B^{\langle s \rangle} = D_2 \cdot S^2 + D_2 \cdot E^{\langle s \rangle} \pmod{Q}$

. In the above decrypted expression $D_2S^2 + D_2E^{\langle s \rangle} \bmod Q$, the term D_2S^2 is okay to be reduced modulo Q, because this term is originally allowed to be reduced modulo Q in the final decryption formula $D_0 + D_1S + D_2S^2 \bmod Q$ as well. However, the problematic term is the noise $D_2 \cdot E^{\langle s \rangle}$, because its coefficients can be any value in $[0, Q-1]$ (since each coefficient of polynomial $D_2 = A^{\langle 1 \rangle} A^{\langle 2 \rangle}$ can be any value in $[0, Q-1]$). Such a huge noise is not allowed for correct final decryption.

To avoid this noise issue, an improved solution is to express the RLWE ciphertext that encrypts D_2S^2 as additions of multiple RLWE ciphertexts with small noises by using the gadget decomposition technique (**??**). For this, we use an RLev ciphertext (**??**) that encrypts S^2. Suppose our gadget vector is $\vec{g} = \left(\frac{Q}{\beta}, \frac{Q}{\beta^2}, \frac{Q}{\beta^3}, \cdots, \frac{Q}{\beta^l}\right)$. Remember that our goal is to find $\mathsf{ct}_\beta = \mathsf{RLWE}_{S,\sigma}(S^2 \cdot D_2)$ given known D_2, unknown S, and known $\mathsf{RLev}^{\beta,l}_{S,\sigma}(S^2) = \left\{\mathsf{RLWE}_{S,\sigma}\left(\frac{Q}{\beta^i} \cdot S\right)\right\}_{i=1}^{l}$. Then, we can derive ct_β as follows:

$\mathsf{ct}_\beta = \mathsf{RLWE}_{S,\sigma}(S^2 \cdot D_2)$

$= \mathsf{RLWE}_{S,\sigma}\left(S^2 \cdot \left(D_{2,1}\frac{Q}{\beta} + D_{2,2}\frac{Q}{\beta^2} + \cdots D_{2,l}\frac{Q}{\beta^l}\right)\right)$ # by decomposing D_2

$= \mathsf{RLWE}_{S,\sigma}\left(S^2 \cdot D_{2,1} \cdot \frac{Q}{\beta}\right) + \mathsf{RLWE}_{S,\sigma}\left(S^2 \cdot D_{2,2} \cdot \frac{Q}{\beta^2}\right) + \cdots + \mathsf{RLWE}_{S,\sigma}\left(S^2 \cdot D_{2,l} \cdot \frac{Q}{\beta^l}\right)$

$= D_{2,1} \cdot \mathsf{RLWE}_{S,\sigma}\left(S^2 \cdot \frac{Q}{\beta}\right) + D_{2,2} \cdot \mathsf{RLWE}_{S,\sigma}\left(S^2 \cdot \frac{Q}{\beta^2}\right) + \cdots + D_{2,l} \cdot \mathsf{RLWE}_{S,\sigma}\left(S^2 \cdot \frac{Q}{\beta^l}\right)$ # where each RLWE ciphertext is an encryption of $S^2\frac{Q}{\beta}, S^2\frac{Q}{\beta^2}, \cdots, S^2\frac{Q}{\beta^l}$ as plaintext with the plaintext scaling factor $\Delta = 1$

$= \langle \mathsf{Decomp}^{\beta,l}(D_2),\ \mathsf{RLev}^{\beta,l}_{S,\sigma}(S^2) \rangle$ # inner product of Decomp and RLev treating them as vectors

If we decrypt the above, we get the following:

$\mathsf{RLWE}^{-1}_{S,\sigma}(\mathsf{ct}_\beta = \langle \mathsf{Decomp}^{\beta,l}(D_2),\ \mathsf{RLev}^{\beta,l}_{S,\sigma}(S^2) \rangle)$ # the scaling factors of $\mathsf{RLev}^{\beta,l}_{S,\sigma}(S^2)$ are all 1

$= D_{2,1} \cdot \left(E'_1 + S^2\frac{Q}{\beta}\right) + D_{2,2} \cdot \left(E'_2 + S^2\frac{Q}{\beta^2}\right) + \cdots + D_{2,l} \cdot \left(E'_l + S^2\frac{Q}{\beta^l}\right)$ # where each E'_i is a noise embedded in $\mathsf{RLWE}_{S,\sigma}\left(S^2 \cdot \frac{Q}{\beta^i}\right)$

$= \sum_{i=1}^{l}(E'_i \cdot D_{2,i}) + S^2 \cdot \left(D_{2,1}\frac{Q}{\beta} + D_{2,2}\frac{Q}{\beta^2} + \cdots + D_{2,l}\frac{Q}{\beta^l}\right)$

$= \sum_{i=1}^{l} \epsilon_i + D_2 \cdot S^2$ # where each $\epsilon_i = E'_i \cdot D_{2,i}$

$\approx D_2 \cdot S^2$ # $\sum_{i=1}^{l} \epsilon_i \ll D_2 \cdot E''$, where E'' is the noise that could've been embedded in $\mathsf{RLWE}_{S,\sigma}(S^2)$

Therefore, we get the following comprehensive relation:
$\mathsf{RLWE}^{-1}_{S,\sigma}(\Delta M^{\langle 1\rangle} + K_1 q) \cdot \mathsf{RLWE}^{-1}_{S,\sigma}(\Delta M^{\langle 2\rangle} + K_2 q) \bmod Q$
$= (\Delta M^{\langle 1\rangle} + E^{\langle 1\rangle} + K_1 q) \cdot (\Delta M^{\langle 2\rangle} + E^{\langle 2\rangle} + K_2 q) \bmod Q$
$= (A^{\langle 1\rangle} \cdot S + B^{\langle 1\rangle}) \cdot (A^{\langle 2\rangle} \cdot S + B^{\langle 2\rangle}) \bmod Q$
$= D_0 + D_1 \cdot S + D_2 \cdot S^2 \bmod Q$ # $D_0 = B^{\langle 1\rangle} B^{\langle 2\rangle}, \quad D_1 = A^{\langle 1\rangle} B^{\langle 2\rangle} + A^{\langle 2\rangle} B^{\langle 1\rangle}, \quad D_2 = A^{\langle 1\rangle} A^{\langle 2\rangle}$
$= \mathsf{RLWE}^{-1}_{S,\sigma}(\mathsf{ct}_\alpha) + \mathsf{RLWE}^{-1}_{S,\sigma}(\mathsf{ct}_\beta) - \sum_{i=1}^{l} (E'_i \cdot D_{2,i}) \bmod Q$
$\mathsf{ct}_\alpha = (D_1, D_0) = (A_\alpha, B_\beta), \quad \mathsf{ct}_\beta = \langle \mathsf{Decomp}^{\beta,l}(D_2), \mathsf{RLev}^{\beta,l}_{S,\sigma}(S^2)\rangle = (A_\beta, B_\beta)$

$= \mathsf{RLWE}^{-1}_{S,\sigma}(\mathsf{ct}_\alpha + \mathsf{ct}_\beta) - \sum_{i=1}^{l} (E'_i \cdot D_{2,i}) \bmod Q$
$= \mathsf{RLWE}^{-1}_{S,\sigma}((A_{\alpha+\beta}, B_{\alpha+\beta})) - \sum_{i=1}^{l} (E'_i \cdot D_{2,i}) \bmod Q$ # $A_{\alpha+\beta} = A_\alpha + A_\beta, \quad B_{\alpha+\beta} = B_\alpha + B_\beta$

From the above, we extract the following relation:
$(\Delta M^{\langle 1\rangle} + E^{\langle 1\rangle} + K_1 q) \cdot (\Delta M^{\langle 2\rangle} + E^{\langle 2\rangle} + K_2 q) \bmod Q$
$= \Delta^2 M^{\langle 1\rangle} M^{\langle 2\rangle} + \Delta \cdot (M^{\langle 1\rangle} E^{\langle 2\rangle} + M^{\langle 2\rangle} E^{\langle 1\rangle}) + q \cdot (\Delta M^{\langle 1\rangle} K_2 + \Delta M^{\langle 2\rangle} K_1 + E^{\langle 1\rangle} K_2 + E^{\langle 2\rangle} K_1) + K_1 K_2 q^2 + E^{\langle 1\rangle} E^{\langle 2\rangle} \bmod Q$
$= \mathsf{RLWE}^{-1}_{S,\sigma}((A_{\alpha+\beta}, B_{\alpha+\beta})) - \sum_{i=1}^{l} (E'_i \cdot D_{2,i}) \bmod Q$

We can re-write the above relation as follows:
$\mathsf{RLWE}^{-1}_{S,\sigma}((A_{\alpha+\beta}, B_{\alpha+\beta})) = A_{\alpha+\beta} \cdot S + B_{\alpha+\beta} \bmod Q$
$= \Delta^2 M^{\langle 1\rangle} M^{\langle 2\rangle} + \Delta \cdot (M^{\langle 1\rangle} E^{\langle 2\rangle} + M^{\langle 2\rangle} E^{\langle 1\rangle}) + q \cdot (\Delta M^{\langle 1\rangle} K_2 + \Delta M^{\langle 2\rangle} K_1 + E^{\langle 1\rangle} K_2 + E^{\langle 2\rangle} K_1) + K_1 K_2 q^2 + E^{\langle 1\rangle} E^{\langle 2\rangle} + \sum_{i=1}^{l} (E'_i \cdot D_{2,i}) \bmod Q$

To verbally interpret the above relation, decrypting the synthetically generated ciphertext $(A_{\alpha+\beta}, B_{\alpha+\beta})$ and applying a reduction modulo Q to it gives us $\Delta^2 M^{\langle 1\rangle} M^{\langle 2\rangle}$ with a lot of noise terms. Meanwhile, as explained in the beginning of this subsection, our goal is to derive a ciphertext whose decryption is $\Delta M^{\langle 1\rangle} M^{\langle 2\rangle}$, also ensuring that the decrypted ciphertext's noise is small enough to be fully eliminated by scaling down the plaintext by Δ at the end. This goal is accomplished by the final rescaling step to be explained in the next subsection.

D-2.7.4 Rescaling

The rescaling step is equivalent to converting the ciphertext $(A_{\alpha+\beta}, B_{\alpha+\beta}) \bmod Q$ into $\left(\left\lceil \frac{A_{\alpha+\beta}}{\Delta} \right\rfloor, \left\lceil \frac{B_{\alpha+\beta}}{\Delta} \right\rfloor\right) \bmod q$, where $\Delta = \left\lfloor \frac{q}{t} \right\rfloor \approx \frac{q}{t}$. The decryption of this rescaled ciphertext (and finally scaling down by Δ) is $\Delta M^{\langle 1\rangle} M^{\langle 2\rangle}$. This is demonstrated below:

$\left\lceil \frac{A_{\alpha+\beta}}{\Delta} \right\rfloor \cdot S + \left\lceil \frac{B_{\alpha+\beta}}{\Delta} \right\rfloor \bmod q$ # decryption of ciphertext $\left(\left\lceil \frac{A_{\alpha+\beta}}{\Delta} \right\rfloor, \left\lceil \frac{B_{\alpha+\beta}}{\Delta} \right\rfloor\right) \bmod q$
$= \left\lceil \frac{A_{\alpha+\beta}}{\Delta} \right\rfloor \cdot S + \left\lceil \frac{B_{\alpha+\beta}}{\Delta} \right\rfloor + K_3 q$ # where $K_3 q$ stands for modulo reduction by q
$= \left\lceil \frac{A_{\alpha+\beta}}{\Delta} \right\rfloor \cdot S + \left\lceil \frac{B_{\alpha+\beta}}{\Delta} \right\rfloor + \frac{K_3 Q}{\Delta}$ # since $Q = \Delta \cdot q$

$$= \left\lceil \frac{1}{\Delta} \cdot (A_{\alpha+\beta} \cdot S + B_{\alpha+\beta} + K_3 Q) \right\rfloor + E_r \ \# \ E_r \text{ is a rounding error}$$

$$= \left\lceil \frac{1}{\Delta} \cdot (A_{\alpha+\beta} \cdot S + B_{\alpha+\beta} \bmod Q) \right\rfloor + E_r$$

$$= \left\lceil \frac{1}{\Delta} \cdot (\Delta^2 M^{\langle 1 \rangle} M^{\langle 2 \rangle} + \Delta \cdot (M^{\langle 1 \rangle} E^{\langle 2 \rangle} + M^{\langle 2 \rangle} E^{\langle 1 \rangle}) + \right.$$

$$\left. q \cdot (\Delta M^{\langle 1 \rangle} K_2 + \Delta M^{\langle 2 \rangle} K_1 + E^{\langle 1 \rangle} K_2 + E^{\langle 2 \rangle} K_1) + K_1 K_2 q^2 + E^{\langle 1 \rangle} E^{\langle 2 \rangle} + \sum_{i=1}^{l} (E'_i \cdot D_{2,i}) \bmod Q) \right\rfloor + E_r$$

as we derived at the end of **??**

$$= \left\lceil \frac{1}{\Delta} \cdot (\Delta^2 M^{\langle 1 \rangle} M^{\langle 2 \rangle} + \Delta \cdot (M^{\langle 1 \rangle} E^{\langle 2 \rangle} + M^{\langle 2 \rangle} E^{\langle 1 \rangle}) + \right.$$

$$\left. q \cdot (\Delta M^{\langle 1 \rangle} K_2 + \Delta M^{\langle 2 \rangle} K_1 + E^{\langle 1 \rangle} K_2 + E^{\langle 2 \rangle} K_1) + K_1 K_2 q^2 + E^{\langle 1 \rangle} E^{\langle 2 \rangle} + \sum_{i=1}^{l} (E'_i \cdot D_{2,i}) + K_4 Q) \right\rfloor + E_r$$

where $K_4 Q$ stands for modulo reduction by Q

$$= \left\lceil \Delta M^{\langle 1 \rangle} M^{\langle 2 \rangle} + (M^{\langle 1 \rangle} E^{\langle 2 \rangle} + M^{\langle 2 \rangle} E^{\langle 1 \rangle}) + q \cdot (M^{\langle 1 \rangle} K_2 + M^{\langle 2 \rangle} K_1) \right.$$

$$\left. + \frac{1}{\Delta} \cdot q \cdot (E^{\langle 1 \rangle} K_2 + E^{\langle 2 \rangle} K_1) + \frac{1}{\Delta} \cdot (K_1 K_2 q^2 + E^{\langle 1 \rangle} E^{\langle 2 \rangle} + \sum_{i=1}^{l} (E'_i \cdot D_{2,i})) + K_5 q \right\rfloor$$

where $K_5 q = K_4 q + M^{\langle 1 \rangle} q K_2 + M^{\langle 2 \rangle} K_1 q$

$$= \left\lceil \Delta M^{\langle 1 \rangle} M^{\langle 2 \rangle} + (M^{\langle 1 \rangle} E^{\langle 2 \rangle} + M^{\langle 2 \rangle} E^{\langle 1 \rangle}) + q \cdot (M^{\langle 1 \rangle} K_2 + M^{\langle 2 \rangle} K_1) + (t + \epsilon) \cdot (E^{\langle 1 \rangle} K_2 + E^{\langle 2 \rangle} K_1) \right.$$

$$\left. + \frac{1}{\Delta} \cdot (K_1 K_2 q^2 + E^{\langle 1 \rangle} E^{\langle 2 \rangle} + \sum_{i=1}^{l} (E'_i \cdot D_{2,i})) + K_5 q \right\rfloor$$

where $\epsilon = \frac{q}{\Delta} - \frac{q}{\frac{q}{t}} = \frac{q}{\left\lfloor \frac{q}{t} \right\rfloor} - \frac{q}{\frac{q}{t}} \approx 0$, thus we substituted $\frac{q}{\Delta} = \epsilon + t$

$$= \left\lceil \Delta M^{\langle 1 \rangle} M^{\langle 2 \rangle} + (M^{\langle 1 \rangle} E^{\langle 2 \rangle} + M^{\langle 2 \rangle} E^{\langle 1 \rangle}) + q \cdot (M^{\langle 1 \rangle} K_2 + M^{\langle 2 \rangle} K_1) + (t + \epsilon) \cdot (E^{\langle 1 \rangle} K_2 + E^{\langle 2 \rangle} K_1) \right.$$

$$\left. + \frac{K_1 K_2 q^2}{\Delta} + \frac{E^{\langle 1 \rangle} E^{\langle 2 \rangle} + \sum_{i=1}^{l} (E'_i \cdot D_{2,i})}{\Delta} + K_5 q \right\rfloor$$

Now, let $\epsilon' = \frac{K_1 K_2 q^2}{\Delta} - \frac{K_1 K_2 q^2}{\frac{q}{t}} = \frac{K_1 K_2 q^2}{\left\lfloor \frac{q}{t} \right\rfloor} - \frac{K_1 K_2 q^2}{\frac{q}{t}} \approx 0.$

Thus, we will substitute $\frac{K_1K_2q^2}{\Delta} = \frac{K_1K_2q^2}{\frac{q}{t}} + \epsilon' = K_1K_2qt + \epsilon'$

$$= \Delta M^{\langle 1\rangle}M^{\langle 2\rangle} + (M^{\langle 1\rangle}E^{\langle 2\rangle} + M^{\langle 2\rangle}E^{\langle 1\rangle}) + q \cdot (M^{\langle 1\rangle}K_2 + M^{\langle 2\rangle}K_1)$$
$$+(t+\epsilon)\cdot(E^{\langle 1\rangle}K_2 + E^{\langle 2\rangle}K_1) + K_1K_2qt + \epsilon' + K_5q + \left\lceil \frac{E^{\langle 1\rangle}E^{\langle 2\rangle} + \sum_{i=1}^{l}(E'_i \cdot D_{2,i})}{\Delta} \right\rfloor$$

$$= \Delta M^{\langle 1\rangle}M^{\langle 2\rangle} + \epsilon'' + K_6q$$
$\#$ where $K_6q = K_5q + q \cdot (M^{\langle 1\rangle}K_2 + M^{\langle 2\rangle}K_1) + K_1K_2qt$,

$$\epsilon'' = M^{\langle 1\rangle}E^{\langle 2\rangle} + M^{\langle 2\rangle}E^{\langle 1\rangle} + (t+\epsilon)\cdot(E^{\langle 1\rangle}K_2 + E^{\langle 2\rangle}K_1) + \left\lceil \frac{E^{\langle 1\rangle}E^{\langle 2\rangle} + \sum_{i=1}^{l}(E'_i \cdot D_{2,i})}{\Delta} \right\rfloor$$

$$= \Delta M^{\langle 1\rangle}M^{\langle 2\rangle} + \epsilon'' \bmod q$$

In conclusion, the ciphertext $\left(\left\lceil \frac{A_{\alpha+\beta}}{\Delta} \right\rfloor, \left\lceil \frac{B_{\alpha+\beta}}{\Delta} \right\rfloor\right) \bmod q$ successfully decrypts to $\Delta M^{\langle 1\rangle}M^{\langle 2\rangle}$ if $\epsilon'' < \frac{\Delta}{2} \approx \frac{q}{2t}$.

Noise Analysis: Among the terms of ϵ'', let's analyze the noise growth of the $(t+\epsilon)\cdot(E^{\langle 1\rangle}K_2 + E^{\langle 2\rangle}K_1)$ term after ciphertext-to-ciphertext multiplication. Each coefficient of K_1 is at most n, because $A^{\langle 1\rangle}\cdot S + B^{\langle 1\rangle} = \Delta M + E^{\langle 1\rangle} + K_1q$, where the maximum possible coefficient value of $A^{\langle 1\rangle}\cdot S + B^{\langle 1\rangle}$ is $q\cdot n$. And the same is true for the coefficients of K_2. Therefore, after scaling down $(t+\epsilon)\cdot(E^{\langle 1\rangle}K_2 + E^{\langle 2\rangle}K_1)$ by Δ upon the final decryption stage, this term's down-scaled noise gets bound by:

$$\frac{1}{\Delta}\cdot(t+\epsilon)\cdot(E^{\langle 1\rangle}K_2 + E^{\langle 2\rangle}K_1) \approx \frac{t}{q}\cdot(t+\epsilon)\cdot(E^{\langle 1\rangle}K_2 + E^{\langle 2\rangle}K_1) < \frac{nt\cdot(t+\epsilon)}{q}\cdot(E^{\langle 1\rangle} + E^{\langle 2\rangle})$$

This implies that for correct decryption, $\frac{nt\cdot(t+\epsilon)}{q}\cdot(E^{\langle 1\rangle} + E^{\langle 2\rangle})$ has to be smaller than 0.5. In other words, $E^{\langle 1\rangle} + E^{\langle 2\rangle}$ has to be smaller than $\frac{q}{2nt\cdot(t+\epsilon)}$. We can do noise analysis for all other terms for ϵ'' in a similar manner. Importantly, upon decryption, the aggregation of all these noise terms' down-scaled values has to be smaller than 0.5 for correct decryption.

Modulus Switch v.s. Rescaling: Notice in the rescaling process, multiplying $\frac{1}{\Delta}$ to $A_{\alpha+\beta}$ and $B_{\alpha+\beta}$ results in two effects: (1) converts $\Delta^2 M^{\langle 1\rangle}M^{\langle 2\rangle}$ into $\Delta M^{\langle 1\rangle}M^{\langle 2\rangle}$; (2) switches the modulus of the *mod-raised* ciphertexts from $Q \to q$. In fact, modulus switch and rescaling are closely equivalent to each other. Modulus switch is a process of changing a ciphertext's modulus (e.g., $q \to q'$), while preserving the property that the decryption of both ciphertexts results in the same plaintext. On the other hand, rescaling refers to the process of changing the scaling factor of a plaintext within a ciphertext (e.g., $\Delta \to \Delta'$). Modulus switch inevitably changes the scaling factor

of the plaintext within the target ciphertext, and rescaling also inevitably changes the modulus of the ciphertext that contains the plaintext (as shown in **??**). Therefore, these two terms can be used interchangeably.

D-2.7.5 Summary

To put all things together, BFV's ciphertext-to-ciphertext multiplication is summarized as follows:

⟨Summary ??⟩ BFV's Ciphertext-to-Ciphertext Multiplication

Suppose we have the following two RLWE ciphertexts:
$\mathsf{RLWE}_{S,\sigma}(\Delta M^{\langle 1\rangle}) = (A^{\langle 1\rangle}, B^{\langle 1\rangle}) \bmod q, \quad \text{where } B^{\langle 1\rangle} = -A^{\langle 1\rangle} \cdot S + \Delta M^{\langle 1\rangle} + E^{\langle 1\rangle} \bmod q$
$\mathsf{RLWE}_{S,\sigma}(\Delta M^{\langle 2\rangle}) = (A^{\langle 2\rangle}, B^{\langle 2\rangle}) \bmod q, \quad \text{where } B^{\langle 2\rangle} = -A^{\langle 2\rangle} \cdot S + \Delta M^{\langle 2\rangle} + E^{\langle 2\rangle} \bmod q$

Multiplication between these two ciphertexts is performed as follows:

1. **ModRaise**
   Forcibly modify the modulus of the ciphertexts $(A^{\langle 1\rangle}, B^{\langle 1\rangle}) \bmod q$ and $(A^{\langle 2\rangle}, B^{\langle 2\rangle}) \bmod q$ to Q (where $Q = q \cdot \Delta$) as follows:
   $(A^{\langle 1\rangle}, B^{\langle 1\rangle}) \bmod Q$
   $(A^{\langle 2\rangle}, B^{\langle 2\rangle}) \bmod Q$

2. **Multiplication**
   Compute the following polynomial multiplications in modulo Q:
   $D_0 = B^{\langle 1\rangle} B^{\langle 2\rangle} \bmod Q$
   $D_1 = B^{\langle 2\rangle} A^{\langle 1\rangle} + B^{\langle 1\rangle} A^{\langle 2\rangle} \bmod Q$
   $D_2 = A^{\langle 1\rangle} \cdot A^{\langle 2\rangle} \bmod Q$

3. **Relinearization**
   Compute the following:
   $\mathsf{ct}_\alpha = (D_1, D_0)$
   $\mathsf{ct}_\beta = \langle \mathsf{Decomp}^{\beta,l}(D_2),\ \mathsf{RLev}^{\beta,l}_{S,\sigma}(S^2)\rangle$.
   $\mathsf{ct}_{\alpha+\beta} = \mathsf{ct}_\alpha + \mathsf{ct}_\beta$

   Then, this property holds: $\mathsf{RLWE}^{-1}_{S,\sigma}(\mathsf{RLWE}_{S,\sigma}(\Delta^2 \cdot M^{\langle 1\rangle} \cdot M^{\langle 2\rangle})) \approx \mathsf{RLWE}^{-1}_{S,\sigma}(\mathsf{ct}_{\alpha+\beta})$

4. **Rescaling**
   Update $\mathsf{ct}_{\alpha+\beta} = (A_{\alpha+\beta}, B_{\alpha+\beta}) \bmod Q$ to $\mathsf{ct'}_{\alpha+\beta} = \left(\left\lceil \frac{A_{\alpha+\beta}}{\Delta} \right\rfloor, \left\lceil \frac{B_{\alpha+\beta}}{\Delta} \right\rfloor\right) \bmod q$.

   This plaintext rescaling process can be also viewed as a modulus switch of the ciphertext $\mathsf{ct}_{\alpha+\beta}$ from $Q \to q$.

Note that after the ciphertext-to-ciphertext multiplication, the plaintext scaling factor $\Delta = \left\lfloor \frac{q}{t} \right\rfloor$, the ciphertext modulus q, the plaintext M, and the private key S stay the same as

before.

The Purpose of ModRaise: When we multiply polynomials at the second step of ciphertext-to-ciphertext multiplication (**??**), the underlying plaintext within the ciphertext temporarily grows to $\Delta^2 M^{\langle 1 \rangle} M^{\langle 2 \rangle}$, which exceeds the allowed maximum boundary q for the plaintext (Summary **??** in **??**). After this point, applying modulo-q reduction to the intermediate result will irrevocably corrupt the plaintext. To avoid the corruption of the plaintext when it grows to $\Delta^2 M^{\langle 1 \rangle} M^{\langle 2 \rangle}$, we temporarily increase the ciphertext modulus from $q \rightarrow Q$, which is sufficiently large to hold $\Delta^2 M^{\langle 1 \rangle} M^{\langle 2 \rangle}$ without wrapping around the boundary of the ciphertext modulus.

Swapping the Order of Relinearization and Rescaling: The order of relinearization and rescaling is interchangeable. Running rescaling before relinearization reduces the size of the ciphertext modulus, and therefore the subsequent relinearization can be executed faster.

D-2.8 Homomorphic Key Switching

BFV's key switching scheme changes an RLWE ciphertext's secret key from S to S'. This scheme is essentially RLWE's key switching scheme with the sign of the $A \cdot S$ term flipped in the encryption and decryption formula. Specifically, this is equivalent to the alternative GLWE version's (**??**) key switching scheme (**??**) with $k = 1$ as follows:

⟨Summary ??⟩ BFV's Key Switching

$$\mathsf{RLWE}_{S',\sigma}(\Delta M) = (0, B) + \langle \mathsf{Decomp}^{\beta,l}(A),\ \mathsf{RLev}^{\beta,l}_{S',\sigma}(S) \rangle$$

D-2.9 Homomorphic Rotation of Input Vector Slots

In this section, we will explain how to homomorphically rotate the elements of an input vector $\vec{v}$ after it is already encoded as a polynomial and encrypted as an RLWE ciphertext. In **??**, we learned how to rotate the coefficients of a polynomial. However, rotating the plaintext polynomial $M(X)$ or RLWE ciphertext polynomials $(A(X), B(X))$ does not necessarily rotate the input vector, which is the source of them.

The key requirement of homomorphic rotation of input vector slots (i.e., input vector) is that this operation should be performed on the RLWE ciphertext such that after this operation, if we decrypt the RLWE ciphertext and decode it, the recovered input vector will be in a rotated state as we expect. We will divide this task into the following two sub-problems:

1. How to indirectly rotate the input vector by updating the plaintext polynomial M to M'?
2. How to indirectly update the plaintext polynomial M to M' by updating the RLWE ciphertext polynomials (A, B) to (A', B')?

D-2.9.1 Rotating Input Vector Slots by Updating the Plaintext Polynomial

In this task, our goal is to modify the plaintext polynomial $M(X)$ such that the first-half elements of the input vector $\vec{v}$ are shifted to the left by h positions in a wrapping manner among them,

and the second-half elements of $\vec{v}$ are also shifted to the left by h positions in a wrapping manner among them. Specifically, if $\vec{v}$ is defined as follows:

$\vec{v} = (v_0, v_1, \cdots, v_{n-1})$

Then, we will denote the h-shifted vector $\vec{v}^{\langle h \rangle}$ as follows:

$$\vec{v}^{\langle h \rangle} = (\underbrace{v_h, v_{h+1}, \cdots, v_0, v_1, \cdots, v_{h-2}, v_{h-1},}_{\text{The first-half } \frac{n}{2} \text{ elements } h\text{-rotated to the left}} \quad \underbrace{v_{\frac{n}{2}+h}, v_{\frac{n}{2}+h+1}, \cdots, v_{\frac{n}{2}+h-2}, v_{\frac{n}{2}+h-1}}_{\text{The second-half } \frac{n}{2} \text{ elements } h\text{-rotated to the left}})$$

Remember from **??** that the BFV encoding scheme's components are as follows:

$\vec{v} = (v_0, v_1, v_2, \cdots, v_{n-1})$ # n-dimensional input vector

$$W = \begin{bmatrix} 1 & 1 & 1 & \cdots & 1 \\ (\omega) & (\omega^3) & (\omega^5) & \cdots & (\omega^{2n-1}) \\ (\omega)^2 & (\omega^3)^2 & (\omega^5)^2 & \cdots & (\omega^{2n-1})^2 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ (\omega)^{n-1} & (\omega^3)^{n-1} & (\omega^5)^{n-1} & \cdots & (\omega^{2n-1})^{n-1} \end{bmatrix}$$

$$= \begin{bmatrix} 1 & 1 & \cdots & 1 & 1 & \cdots & 1 & 1 \\ (\omega) & (\omega^3) & \cdots & (\omega^{\frac{n}{2}-1}) & (\omega^{-(\frac{n}{2}-1)}) & \cdots & (\omega^{-3}) & (\omega^{-1}) \\ (\omega)^2 & (\omega^3)^2 & \cdots & (\omega^{\frac{n}{2}-1})^2 & (\omega^{-(\frac{n}{2}-1)})^2 & \cdots & (\omega^{-3})^2 & (\omega^{-1})^2 \\ \vdots & \vdots & \vdots & \ddots & \vdots & & & \\ (\omega)^{n-1} & (\omega^3)^{n-1} & \cdots & (\omega^{\frac{n}{2}-1})^{n-1} & (\omega^{-(\frac{n}{2}-1)})^{n-1} & \cdots & (\omega^{-3})^{n-1} & (\omega^{-1})^{n-1} \end{bmatrix}$$

, where $\omega = g^{\frac{t-1}{2n}} \bmod t$ (g is a generator of $\mathbb{Z}_t^\times$)

The encoding matrix that converts $\vec{v}$ into $\vec{m}$ (i.e., n coefficients of the plaintext polynomial $M(X)$)

$\Delta\vec{m} = n^{-1} \cdot \Delta W \cdot I_n^R \cdot \vec{v}$

A vector containing the scaled n integer coefficients of the plaintext polynomial

$\Delta M = \sum_{i=0}^{n-1} (\Delta m_i X^i)$

The integer polynomial that isomorphically encodes the input vector $\vec{v}$

We learned from **??** that decoding the polynomial $M(X)$ back to $\vec{v}$ is equivalent to evaluating $M(X)$ at the following n distinct primitive ($\mu = 2n$)-th root of unity: $\{\omega, \omega^3, \omega^5, \cdots, \omega^{2n-3}, \omega^{2n-1}\}$. Thus, the above decoding process is equivalent to the following:

$\vec{v} = \frac{W^T \Delta\vec{m}}{\Delta} = (W_0^T \cdot \vec{m},\ W_1^T \cdot \vec{m},\ W_2^T \cdot \vec{m},\ \cdots,\ W_{n-1}^T \cdot \vec{m})$

$= (\ M(\omega),\ M(\omega^3),\ M(\omega^5), \cdots, M(\omega^{2n-3}),\ M(\omega^{2n-1})\ )$

$= (\ M(\omega),\ M(\omega^3),\ M(\omega^5), \cdots, M(\omega^{n-3}),\ M(\omega^{n-1}),\ M(\omega^{-(n-1)}),\ M(\omega^{-(n-3)}), \cdots, M(\omega^{-3}),\ M(\omega{-}1)\ )$

Now, our next task is to modify $M(X)$ to $M'(X)$ such that decoding $M'(X)$ will give us a modified input vector $\vec{v}^{\langle h \rangle}$ that is a rotation of the first half elements of $\vec{v}$ by h positions to the left (in a wrapping manner among them), and the second half elements of it also rotated by h positions

to the left (in a wrapping manner among them). To accomplish this rotation, we will take a 2-step solution:

1. To convert $M(X)$ into $M'(X)$, we will define the new mapping σ_M as follows:
   $\sigma_M : (M(X), h) \in (\mathcal{R}_{\langle n,t \rangle}, \mathbb{Z}_n) \longrightarrow M'(X) \in \mathcal{R}_{\langle n,t \rangle}$
   , where h is the number of rotation positions to be applied to $\vec{v}$.
2. To decode $M'(X)$ into the rotated input vector $\vec{v}^{\langle h \rangle}$, we need to re-design our decoding scheme by modifying $\mathsf{Encoding}_1$'s (**??**) isomorphic mapping $\sigma : M(X) \in \mathcal{R}_{\langle n,t \rangle} \longrightarrow \vec{v} \in \mathbb{Z}^n$

Converting $M(X)$ into $M'(X)$: Our first task is to convert $M(X)$ into $M'(X)$, which is equivalent to applying our new mapping $\sigma_M : (M(X), h) \in (\mathcal{R}_{\langle n,t \rangle}, \mathbb{Z}_n) \longrightarrow M'(X) \in \mathcal{R}_{\langle n,t \rangle}$, such that decoding $M'(X)$ gives a rotated input vector $\vec{v}^{\langle h \rangle}$ whose first half of the elements in $\vec{v}$ are rotated by h positions to the left (in a wrapping manner among them), and the second half of the elements are also rotated by h positions to the left (in a wrapping manner among them). To design σ_M that satisfies this requirement, we will use the number 5^j which has the following two special properties (based on number theory):

- $(5^j \bmod 2n)$ and $(-5^j \bmod 2n)$ generate all odd numbers between $[0, 2n)$ for the integer j where $0 \leq j < \frac{n}{2}$.
- For each integer j where $0 \leq j < \frac{n}{2}$, $(5^j \bmod 2n) + (-5^j \bmod 2n) \equiv 0 \bmod 2n$.

For example, suppose the modulus $2n = 16$. Then,

$5^0 \bmod 16 = 1$ $\quad$ $-(5)^0 \bmod 16 = 15$
$5^1 \bmod 16 = 5$ $\quad$ $-(5)^1 \bmod 16 = 11$
$5^2 \bmod 16 = 9$ $\quad$ $-(5)^2 \bmod 16 = 7$
$5^3 \bmod 16 = 13$ $\quad$ $-(5)^3 \bmod 16 = 3$

As shown above, $0 \leq j < 4$ generate all odd numbers between $[0, 16)$. Also, for each j in $0 \leq j < 4$, $(5^j \bmod 16) + (-5^j \bmod 16) = 16$.

Let's define $J(h) = 5^h \bmod 2n$, and $J_*(h) = -5^h \bmod 2n$. Based on $J(h)$ and $J_*(h)$, we will define the mapping $\sigma_M : M(X) \to M'(X)$ as follows:
$\sigma_M : M(X) \to M(X^{J(h)})$

Given a plaintext polynomial $M(X)$, in order to give its decoded version of input vector $\vec{v}$ the effect of the first half of the elements being rotated by h positions to the left (in a wrapping manner) and the second half of the elements also being rotated by h positions to the left (in a wrapping manner), we update the current plaintext polynomial $M(X)$ to a new polynomial $M'(X) = M(X^{J(h)}) = M(X^{5^h})$ by applying the σ_M mapping, where h is the number of positions for left rotations for the first half and second half of the elements of $\vec{v}$.

Decoding $M'(X)$ into $\vec{v}^{\langle h \rangle}$: Our second task is to modify our original decoding scheme in order to successfully decode $M'(X)$ into the rotated input vector $\vec{v}^{\langle h \rangle}$. For this, we will modify our original isomorphic mapping $\sigma : M(X) \longrightarrow \vec{v}$, from:
$\sigma : M(X) \in \mathcal{R}_{\langle n,q \rangle} \longrightarrow \big(M(\omega), M(\omega^3), M(\omega^5), \cdots, M(\omega^{2n-1})\big) \in \mathbb{Z}^n$ # designed in **??**

, to the following:

$$\sigma_J : M(X) \in \mathcal{R}_{\langle n,t \rangle} \longrightarrow (M(\omega^{J(0)}), M(\omega^{J(1)}), M(\omega^{J(2)}), \cdots, M(\omega^{J(\frac{n}{2}-1)}), \\ M(\omega^{J_*(0)}),\ M(\omega^{J_*(1)}),\ M(\omega^{J_*(2)}), \cdots, M(\omega^{J_*(\frac{n}{2}-1)})\ ) \in \mathbb{Z}^n$$

The common aspect between σ and σ_J is that they both evaluate the polynomial $M(X)$ at n distinct primitive ($\mu = 2n$)-th roots of unity (i.e., w^i for all odd i between $[0, 2n]$). In the case of the σ_J mapping, note that $J(j) = 5^j \bmod 2n \quad$ and $\quad J_*(j) = -5^j \bmod 2n \quad$ for each j in $0 \leq j < \dfrac{n}{2}$ cover all odd numbers between $[0, 2n]$. Therefore, $\omega^{J(j)}$ and $\omega^{J_*(j)}$ between $0 \leq j < \frac{n}{2}$ cover all n distinct primitive ($\mu = 2n$)-th roots of unity.

Meanwhile, the difference between σ and σ_J is the order of the output vector elements. In the σ mapping, the order of evaluated coordinates for $M(X)$ is $\omega, \omega^3, \cdots, \omega^{2n-1}$, whereas in the σ_J mapping, the order of evaluated coordinates is $\omega^{J(0)}, \omega^{J(1)}, \cdots, \omega^{J(\frac{n}{2}-1)}, \omega^{J_*(0)}, \omega^{J_*(1)}, \cdots, \omega^{J_*(\frac{n}{2}-1)}$. We will later explain why we modified the ordering like this.

In the original $\mathsf{Decoding}_2$ process (**??**), applying the σ mapping to a plaintext polynomial $M(X)$ was equivalent to computing the following:

$\vec{v'} = (\ M(\omega),\ M(\omega^3),\ M(\omega^5), \cdots, M(\omega^{2n-3}),\ M(\omega^{2n-1}))$

$= (\ M(\omega),\ M(\omega^3),\ M(\omega^5), \cdots, M(\omega^{n-1}),\ M(\omega^{-(n-1)}), \cdots,\ M(\omega^{-3}),\ M(\omega^{-1})\ )$

$= (\ W_0^T \cdot \vec{m},\ W_1^T \cdot \vec{m},\ W_2^T \cdot \vec{m},\ \cdots,\ W_{n-1}^T \cdot \vec{m}\ )$ # where W_i^T is the $(i+1)$-th row of W^T

$= W^T \vec{m}$

Similarly, the modified σ_J mapping to the plaintext polynomial $M(X)$ is equivalent to computing the following:

$\vec{v'} = (\ M(\omega^{J(0)}),\ M(\omega^{J(1)}),\ M(\omega^{J(2)}), \cdots, M(\omega^{J(\frac{n}{2}-1)}),\ M(\omega^{J_*(0)}),\ M(\omega^{J_*(1)}), \cdots, M(\omega^{J_*(\frac{n}{2}-1)})\ )$

$= (\ \tilde{W}_0^* \cdot \vec{m},\ \tilde{W}_1^* \cdot \vec{m},\ \tilde{W}_2^* \cdot \vec{m},\ \cdots,\ \tilde{W}_{n-1}^* \cdot \vec{m}\ )$

$= \tilde{W}^* \vec{m}$, where

$$\tilde{W}^* = \begin{bmatrix} 1 & (\omega^{J(0)}) & (\omega^{J(0)})^2 & \cdots & (\omega^{J(0)})^{n-1} \\ 1 & (\omega^{J(1)}) & (\omega^{J(1)})^2 & \cdots & (\omega^{J(1)})^{n-1} \\ 1 & (\omega^{J(2)}) & (\omega^{J(2)})^2 & \cdots & (\omega^{J(2)})^{n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & (\omega^{J(\frac{n}{2}-1)}) & (\omega^{J(\frac{n}{2}-1)})^2 & \cdots & (\omega^{J(\frac{n}{2}-1)})^{n-1} \\ 1 & (\omega^{J_*(0)}) & (\omega^{J_*(0)})^2 & \cdots & (\omega^{J_*(0)})^{n-1} \\ 1 & (\omega^{J_*(1)}) & (\omega^{J_*(1)})^2 & \cdots & (\omega^{J_*(1)})^{n-1} \\ 1 & (\omega^{J_*(2)}) & (\omega^{J_*(2)})^2 & \cdots & (\omega^{J_*(2)})^{n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & (\omega^{J_*(\frac{n}{2}-1)}) & (\omega^{J_*(\frac{n}{2}-1)})^2 & \cdots & (\omega^{J_*(\frac{n}{2}-1)})^{n-1} \end{bmatrix}$$

$$\tilde{W} = \begin{bmatrix} 1 & 1 & \cdots & 1 & 1 & 1 & \cdots & 1 \\ (\omega^{J(\frac{n}{2}-1)}) & (\omega^{J(\frac{n}{2}-2)}) & \cdots & (\omega^{J(0)}) & (\omega^{J_*(\frac{n}{2}-1)}) & (\omega^{J_*(\frac{n}{2}-2)}) & \cdots & (\omega^{J_*(0)}) \\ (\omega^{J(\frac{n}{2}-1)})^2 & (\omega^{J(\frac{n}{2}-2)})^2 & \cdots & (\omega^{J(0)})^2 & (\omega^{J_*(\frac{n}{2}-1)})^2 & (\omega^{J_*(\frac{n}{2}-2)})^2 & \cdots & (\omega^{J_*(0)})^2 \\ \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \vdots & \vdots \\ (\omega^{J(\frac{n}{2}-1)})^{n-1} & (\omega^{J(\frac{n}{2}-2)})^{n-1} & \cdots & (\omega^{J(0)})^{n-1} & (\omega^{J_*(\frac{n}{2}-1)})^{n-1} & (\omega^{J_*(\frac{n}{2}-2)})^{n-1} & \cdots & (\omega^{J_*(0)})^{n-1} \end{bmatrix}$$

From this point, we will replace W in the $\mathsf{Encoding}_1$ process (**??**) by $\tilde{W}$, and W^T in the $\mathsf{Decoding}_2$ process by $\tilde{W}^*$.

To demonstrate that $\tilde{W}$ is a valid encoding matrix like W and $\tilde{W}^*$ is a valid decoding matrix like W^T, we need to prove the following 2 aspects:

- $\boldsymbol{\tilde{W}}$ **is a basis of the** $\boldsymbol{n}$**-dimensional vector space:** This is true, because $\tilde{W}$ is simply a row-wise re-ordering of W, which is still a basis of the $\boldsymbol{n}$-dimensional vector space.
- $\boldsymbol{\tilde{W}^* \cdot \tilde{W} = n \cdot I_n^R}$ (to satisfy Theorem **??** in **??**): This proof is split into 2 sub-proofs:
  1. $\boldsymbol{\tilde{W}^* \cdot \tilde{W}}$ has value $\boldsymbol{n}$ along the anti-diagonal line: Each element along the anti-diagonal line of $\boldsymbol{\tilde{W}^* \cdot \tilde{W}}$ is computed as $\boldsymbol{\sum_{i=0}^{n-1} \omega^{2nik} = n}$ where $\boldsymbol{k}$ is some integer.
  2. $\boldsymbol{\tilde{W}^* \cdot \tilde{W}}$ has value 0 at all other coordinates: All other elements except for the ones along the anti-diagonal lines are $\boldsymbol{\sum_{i=0}^{n-1} \omega^{2i} \frac{\omega^n - 1}{\omega - 1} = 0}$ (by Geometric Sum).

  We provide the Python script that empirically demonstrates this.

Therefore, $\tilde{W}^*$ and $\tilde{W}$ are valid encoding & decoding matrices that transform $\vec{v}$ into $M(X)$.

Now, let's think about what will be the structure of $\vec{v}^{\langle h \rangle}$ (i.e., the first-half elements of $\vec{v}$ being rotated h positions to the left in a wrapping manner among them and the second-half elements of it also being rotated h positions to the left in a wrapping manner among them). Remember that $\vec{v}$ is as follows:

$$\begin{aligned} \vec{v} &= (\ M(\omega^{J(0)}),\ M(\omega^{J(1)}), \cdots, M(\omega^{J(\frac{n}{2}-1)}),\ M(\omega^{J_*(0)}),\ M(\omega^{J_*(1)}), \cdots, M(\omega^{J_*(\frac{n}{2}-1)})) \\ &= (\ \tilde{W}_0^* \cdot \vec{m},\ \tilde{W}_1^* \cdot \vec{m},\ \tilde{W}_2^* \cdot \vec{m},\ \cdots,\ \tilde{W}_{n-1}^* \cdot \vec{m}\ ) \end{aligned}$$

Thus, the state of $\vec{v}^{\langle h \rangle}$ which is equivalent to rotating $\vec{v}$ by h positions to the left for the first-half and second-half element groups will be the following:

$$\begin{aligned} \vec{v}^{\langle h \rangle} = (\ & M(\omega^{J(h)}),\ M(\omega^{J(h+1)}), \cdots, M(\omega^{J(\frac{n}{2}-1)}),\ M(\omega^{J(0)}),\ M(\omega^{J(1)}), \cdots,\ M(\omega^{J(h-2)}),\ M(\omega^{J(h-1)}), \\ & M(\omega^{J_*(h)}),\ M(\omega^{J_*(h+1)}), \cdots, M(\omega^{J_*(\frac{n}{2}-1)}),\ M(\omega^{J_*(0)}),\ M(\omega^{J_*(1)}), \cdots,\ M(\omega^{J_*(h-2)}),\ M(\omega^{J_*(h-1)})\ ) \end{aligned}$$

Notice that the above computation of $\vec{v}^{\langle h \rangle}$ is equivalent to vertically rotating the upper $\frac{n}{2}$ rows of $\tilde{W}^*$ by h positions upward (in a wrapping manner among them), rotating the lower $\frac{n}{2}$ rows of $\tilde{W}^*$ by h positions upward (in a wrapping manner among them), and multiplying the resulting matrix with $\vec{m}$. However, it is not desirable to directly modify the decoding matrix $\tilde{W}^*$ like this in practice, because then the decoding matrix loses its consistency. Therefore, instead of directly modifying $\tilde{W}^*$, we will modify $\vec{m}$ to $\vec{m}'$ (i.e., modify $M(X)$ to $M'(X)$) such that the relation $\vec{v}^{\langle h \rangle} = \tilde{W}^* \cdot \vec{m}'$ holds. Let's extract the upper-half rows of $\tilde{W}^*$ and denote this $\frac{n}{2} \times n$ matrix as $\tilde{H}_1^*$. Then, $\tilde{H}_1^*$ is equivalent to a Vandermonde matrix (Definition **??** in **??**) in the form of $V(\omega^{J(0)}, \omega^{J(1)}, \cdots, \omega^{J(\frac{n}{2}-1)})$. Similarly, let's extract the lower-half rows of $\tilde{W}^*$ and denote this

$\frac{n}{2} \times n$ matrix as $\tilde{H}_2^*$. Then, $\tilde{H}_2^*$ is equivalent to a Vandermonde matrix (Definition **??** in **??**) in the form of $V(\omega^{J_*(0)}, \omega^{J_*(1)}, \cdots, \omega^{J_*(\frac{n}{2}-1)})$.

Now, let's vertically rotate the rows of $\tilde{H}_1^*$ by h positions upward and denote it as $\tilde{H}_1^{*\langle h \rangle}$; and vertically rotate the rows of $\tilde{H}_2^*$ by h positions upward and denote it as $\tilde{H}_2^{*\langle h \rangle}$. And let's denote the $\frac{n}{2}$-dimensional vector comprising the first-half elements of $\vec{v}^{\langle h \rangle}$ as $\vec{v}_1^{\langle h \rangle}$, and the $\frac{n}{2}$-dimensional vector comprising the second-half elements of $\vec{v}^{\langle h \rangle}$ as $\vec{v}_2^{\langle h \rangle}$. Then, computing (i.e., decoding) $\vec{v}_1^{\langle h \rangle} = \tilde{H}_1^{*\langle h \rangle} \cdot \vec{m}$ is equivalent to modifying $M(X)$ to $M'(X) = M(X^{J(h)})$ (whose coefficient vector is $\vec{m}^{\langle h \rangle}$) and then computing (i.e., decoding) $\vec{v}_1^{\langle h \rangle} = \tilde{H}_1^* \cdot \vec{m}^{\langle h \rangle}$. This is because:

$\vec{v}_1^{\langle h \rangle} = \tilde{H}_1^{*\langle h \rangle} \cdot \vec{m}$

$= (\ \tilde{W}_h^* \cdot \vec{m},\ \tilde{W}_{h+1}^* \cdot \vec{m},\ \tilde{W}_{h+2}^* \cdot \vec{m}, \cdots, \tilde{W}_{\frac{n}{2}-1}^* \cdot \vec{m},\ \tilde{W}_0^* \cdot \vec{m},\ \tilde{W}_1^* \cdot \vec{m}, \cdots,\ \tilde{W}_{h-2}^* \cdot \vec{m},\ \tilde{W}_{h-1}^* \cdot \vec{m})$

$= (\ M((\omega^{J(h)})^{J(0)}),\ M((\omega^{J(h)})^{J(1)}),\ M((\omega^{J(h)})^{J(2)}), \cdots, M((\omega^{J(h)})^{J(\frac{n}{2}-1)})\ )$

\# This is equivalent to evaluating the polynomial $M(X^{J(h)})$ at the following $\frac{n}{2}$ distinct $(\mu = 2n)$-th roots of unity: $\omega^{J(0)}, \omega^{J(1)}, \omega^{J(2)}, \cdots, \omega^{J(\frac{n}{2}-1)}$

$= (\ M(\omega^{J(h)\cdot J(0)}),\ M(\omega^{J(h)\cdot J(1)}),\ M(\omega^{J(h)\cdot J(2)}), \cdots, M(\omega^{J(h)\cdot J(\frac{n}{2}-1)})\ )$

$= (\ M(\omega^{5^h \cdot 5^0}),\ M(\omega^{5^h \cdot 5^1}),\ M(\omega^{5^h \cdot 5^2}), \cdots, M(\omega^{5^h \cdot 5^{n/2-1}})\ )$

$= (\ M(\omega^{5^h}),\ M(\omega^{5^{h+1}}),\ M(\omega^{5^{h+2}}), \cdots, M(\omega^{5^{h+n/2-1}})\ )$ \# note that $5^{\frac{n}{2}} \bmod 2n = 1$

$= (\ M(\omega^{J(h)}),\ M(\omega^{J(h+1)}),\ M(\omega^{J(h+2)}), \cdots, M(\omega^{J(\frac{n}{2}-1)}),\ M(\omega^{J(0)}),\ M(\omega^{J(1)}),$
$\cdots,\ M(\omega^{J(h-2)}),\ M(\omega^{J(h-1)}))$

$= \tilde{H}_1^* \cdot \vec{m}^{\langle h \rangle}$ \# where $\vec{m}^{\langle h \rangle}$ contains the n coefficients of the polynomial $M(X^{J(h)})$

Similarly, computing (i.e., decoding) $\vec{v}_2^{\langle h \rangle} = \tilde{H}_2^{*\langle h \rangle} \cdot \vec{m}$ is equivalent to modifying $M(X)$ to $M'(X) = M(X^{J(h)})$ (whose coefficient vector is $\vec{m}^{\langle h \rangle}$) and then computing (i.e., decoding) $\vec{v}_2^{\langle h \rangle} = \tilde{H}_2^* \cdot \vec{m}^{\langle h \rangle}$. This is because,

$\vec{v}_2^{\langle h \rangle} = \tilde{H}_2^{*\langle h \rangle} \cdot \vec{m}$

$= (\ \tilde{W}_{\frac{n}{2}+h}^* \cdot \vec{m},\ \tilde{W}_{\frac{n}{2}+h+1}^* \cdot \vec{m},\ \tilde{W}_{\frac{n}{2}+h+2}^* \cdot \vec{m}, \cdots, \tilde{W}_{n-1}^* \cdot \vec{m},\ \tilde{W}_{\frac{n}{2}}^* \cdot \vec{m},\ \tilde{W}_{\frac{n}{2}+1}^* \cdot \vec{m}, \cdots,$
$\tilde{W}_{\frac{n}{2}+h-2}^* \cdot \vec{m},\ \tilde{W}_{\frac{n}{2}+h-1}^* \cdot \vec{m})$

$= (\ M((\omega^{J(h)})^{J_*(0)}),\ M((\omega^{J(h)})^{J_*(1)}),\ M((\omega^{J(h)})^{J_*(2)}), \cdots, M((\omega^{J(h)})^{J_*(\frac{n}{2}-1)})\ )$

\# This is equivalent to evaluating the polynomial $M(X^{J(h)})$ at the following $\frac{n}{2}$ distinct $(\mu = 2n)$-th roots of unity: $\omega^{J_*(0)}, \omega^{J_*(1)}, \omega^{J_*(2)}, \cdots, \omega^{J_*(\frac{n}{2}-1)}$

$= (\ M(\omega^{J(h)\cdot J_*(0)}),\ M(\omega^{J(h)\cdot J_*(1)}),\ M(\omega^{J(h)\cdot J_*(2)}), \cdots, M(\omega^{J(h)\cdot J_*(\frac{n}{2}-1)})\ )$

$= (\ M(\omega^{5^h \cdot -5^0}),\ M(\omega^{5^h \cdot -5^1}),\ M(\omega^{5^h \cdot -5^2}), \cdots, M(\omega^{5^h \cdot -5^{n/2-1}})\ )$

$= (\ M(\omega^{-5^h}),\ M(\omega^{-5^{h+1}}),\ M(\omega^{-5^{h+2}}), \cdots, M(\omega^{-5^{h+n/2-1}})\ )$ \# note that $-(5^{\frac{n}{2}} \bmod 2n) = -1$

$= (\ M(\omega^{J_*(h)}),\ M(\omega^{J_*(h+1)}),\ M(\omega^{J_*(h+2)}), \cdots, M(\omega^{J_*(\frac{n}{2}-1)}),\ M(\omega^{J_*(0)}),\ M(\omega^{J_*(1)}),$
$\cdots,\ M(\omega^{J_*(h-2)}),\ M(\omega^{J_*(h-1)}))$

$= \tilde{H}_2^* \cdot \vec{m}^{\langle h \rangle}$

The above derivations demonstrate that $\vec{v}_1^{\langle h \rangle} = \tilde{H}_1^* \cdot \vec{m}^{\langle h \rangle}$, and $\vec{v}_2^{\langle h \rangle} = \tilde{H}_2^* \cdot \vec{m}^{\langle h \rangle}$. Combining these two findings, we reach the conclusion that $\vec{v}^{\langle h \rangle} = \tilde{W}^* \cdot \vec{m}^{\langle h \rangle}$: rotating the first-half elements of the input vector $\vec{v}$ by h positions to the left and the second-half elements of it by h positions also to the left is equivalent to updating the plaintext polynomial $M(X)$ to $M(X^{J(h)})$ and then decoding it with the decoding matrix $\tilde{W}^*$.

However, now a new problem is that we cannot directly update the plaintext $M(X)$ to $M(X^{J(h)})$, because $M(X)$ is encrypted as an RLWE ciphertext. Therefore, we need to instead update the RLWE ciphertext components (A, B) to *indirectly* by updating $M(X)$ to $M(X^{J(X)})$. We will explain this in the next subsection.

D-2.9.2 Updating the Plaintext Polynomial by Updating the Ciphertext Polynomials

Given an RLWE ciphertext $\mathsf{ct} = (A, B)$, our goal is to update $\mathsf{ct} = (A, B)$ to $C^{\langle h \rangle} = (A^{\langle h \rangle}, B^{\langle h \rangle})$ such that decrypting it gives the input vector $\vec{v}^{\langle h \rangle}$. That is, the following relation should hold:
$\mathsf{RLWE}_{S,\sigma}^{-1}(\ C^{\langle h \rangle} = (A^{\langle h \rangle}, B^{\langle h \rangle})\ ) = \Delta M(X^{J(h)}) + E'$

Remember that in the RLWE cryptosystem (**??**)'s alternative version (**??**), the plaintext and ciphertext pair have the following relation:
$\Delta M(X) + E(X) = A(X) \cdot S(X) + B(X) \approx \Delta M(X)$

If we apply $X = X^{J(h)}$ in the above relation, we can derive the following relation:
$\Delta M(X^{J(h)}) + E(X^{J(h)}) = A(X^{J(h)}) \cdot S(X^{J(h)}) + B(X^{J(h)}) \approx \Delta M(X^{J(h)})$

This relation implies that if we decrypt the ciphertext $C^{\langle h \rangle} = (A(X^{J(h)}), B(X^{J(h)}))$ with $S(X^{J(h)})$ as the secret key, then we get $\Delta M(X^{J(h)})$. Therefore, $C^{\langle h \rangle} = (A(X^{J(h)}), B(X^{J(h)}))$ is the RLWE ciphertext we are looking for, because decrypting it and then decoding its plaintext $M(X^{J(h)})$ will give us the input vector $\vec{v}^{\langle h \rangle}$.

We can easily convert $\mathsf{ct} = (A(X), B(X))$ into $C^{\langle h \rangle} = (A(X^{J(h)}), B(X^{J(h)}))$ by applying $X^{J(h)}$ to X for each of the $A(X)$ and $B(X)$ polynomials. However, after that, notice that the decryption key of the RLWE ciphertext $C^{\langle h \rangle} = (A(X^{J(h)}), B(X^{J(h)}))$ has been changed from $S(X)$ to $S(X^{J(h)})$. Thus, we need to additionally switch the ciphertext $C^{\langle h \rangle}$'s key from $S(X^{J(h)}) \to S(X)$, which is equivalent to converting $\mathsf{RLWE}_{S(X^{J(h)}),\sigma}(C^{\langle h \rangle} = (A(X^{J(h)}), B(X^{J(h)})))$ into $\mathsf{RLWE}_{S,\sigma}(C^{\langle h \rangle} = (A(X^{J(h)}), B(X^{J(h)})))$. For this, we will apply the BFV key switching technique (Summary **??**) learned in **??** as follows:

$$\underbrace{\mathsf{RLWE}_{S(X),\sigma}(\Delta M(X^{J(h)}))}_{\substack{\text{the result of}\\ \text{plaintext-to-ciphertext addition}}} = \underbrace{(\ 0, B(X^{J(h)})\ )}_{\substack{\text{the plaintext } B(X^{J(h)})\\ \text{(trivial ciphertext)}}} + \langle\ \underbrace{\mathsf{Decomp}^{\beta,l}(A(X^{J(h)})),\ \mathsf{RLev}_{S(X),\sigma}^{\beta,l}(S(X^{J(h)}))}_{\substack{\text{an RLWE ciphertext encrypting } A(X^{J(h)}) \cdot S(X^{J(h)})\\ \text{which is key-switched from } S(X^{J(h)}) \to S(X)}}\ \rangle$$

D-2.9.3 Summary

We summarize the procedure of rotating the BFV input vectors as follows:

⟨Summary ??⟩ BFV's Homomorphic Rotation of Input Vector Slots

To support input vector slot rotation, we update the original encoding matrix in $\mathsf{Encoding}_1$ as follows:

$$\tilde{W} = \begin{bmatrix} 1 & 1 & \cdots & 1 & 1 & 1 & \cdots & 1 \\ (\omega^{J(\frac{n}{2}-1)}) & (\omega^{J(\frac{n}{2}-2)}) & \cdots & (\omega^{J(0)}) & (\omega^{J_*(\frac{n}{2}-1)}) & (\omega^{J_*(\frac{n}{2}-2)}) & \cdots & (\omega^{J_*(0)}) \\ (\omega^{J(\frac{n}{2}-1)})^2 & (\omega^{J(\frac{n}{2}-2)})^2 & \cdots & (\omega^{J(0)})^2 & (\omega^{J_*(\frac{n}{2}-1)})^2 & (\omega^{J_*(\frac{n}{2}-2)})^2 & \cdots & (\omega^{J_*(0)})^2 \\ \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \vdots \\ (\omega^{J(\frac{n}{2}-1)})^{n-1} & (\omega^{J(\frac{n}{2}-2)})^{n-1} & \cdots & (\omega^{J(0)})^{n-1} & (\omega^{J_*(\frac{n}{2}-1)})^{n-1} & (\omega^{J_*(\frac{n}{2}-2)})^{n-1} & \cdots & (\omega^{J_*(0)})^{n-1} \end{bmatrix}$$

, and update the original decoding matrix in $\mathsf{Decoding}_2$ as follows:

$$\tilde{W}^* = \begin{bmatrix} 1 & (\omega^{J(0)}) & (\omega^{J(0)})^2 & \cdots & (\omega^{J(0)})^{n-1} \\ 1 & (\omega^{J(1)}) & (\omega^{J(1)})^2 & \cdots & (\omega^{J(1)})^{n-1} \\ 1 & (\omega^{J(2)}) & (\omega^{J(2)})^2 & \cdots & (\omega^{J(2)})^{n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & (\omega^{J(\frac{n}{2}-1)}) & (\omega^{J(\frac{n}{2}-1)})^2 & \cdots & (\omega^{J(\frac{n}{2}-1)})^{n-1} \\ 1 & (\omega^{J_*(0)}) & (\omega^{J_*(0)})^2 & \cdots & (\omega^{J_*(0)})^{n-1} \\ 1 & (\omega^{J_*(1)}) & (\omega^{J_*(1)})^2 & \cdots & (\omega^{J_*(1)})^{n-1} \\ 1 & (\omega^{J_*(2)}) & (\omega^{J_*(2)})^2 & \cdots & (\omega^{J_*(2)})^{n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & (\omega^{J_*(\frac{n}{2}-1)}) & (\omega^{J_*(\frac{n}{2}-1)})^2 & \cdots & (\omega^{J_*(\frac{n}{2}-1)})^{n-1} \end{bmatrix}$$

, where $J(h)$ is the *rotation helper formula*: $J(h) = 5^h \bmod 2n, \quad J_*(h) = -5^h \bmod 2n$

Suppose we have an RLWE ciphertext and a key-switching key as follows:
$\mathsf{RLWE}_{S,\sigma}(\Delta M) = (A, B), \quad \mathsf{RLev}^{\beta,l}_{S,\sigma}(S(X^{J(h)}))$

Then, the procedure of rotating the first-half elements of the ciphertext's original input vector $\vec{v}$ by h positions to the left (in a wrapping manner among them) and the second-half elements of $\vec{v}$ by h positions to the left (in a wrapping manner among them) is as follows:
1. Update $A(X)$, $B(X)$ to $A(X^{J(h)})$, $B(X^{J(h)})$.
2. Perform the following key switching (**??**) from $S(X^{J(h)})$ to $S(X)$:
   $\mathsf{RLWE}_{S(X),\sigma}(\Delta M(X^{J(h)})) = (0, B(X^{J(h)})) + \langle \mathsf{Decomp}^{\beta,l}(A(X^{J(h)})),\ \mathsf{RLev}^{\beta,l}_{S(X),\sigma}(S(X^{J(h)})) \rangle$

D-2.9.4 Encoding Example

Suppose we have the following setup:

$\mu = 8, n = \dfrac{\mu}{2} = 4,\ t = 17,\ q = 2^6 = 64,\ n^{-1} = 13,\ \Delta = 2$

$\mathcal{R}_{\langle 4,17 \rangle} = \mathbb{Z}_{17}[X]/(X^4 + 1)$

The roots of $X^4 + 1 \pmod{17}$ are $X = \{2, 8, 15, 9\}$, as demonstrated as follows:

$2^4 \equiv 8^4 \equiv 15^4 \equiv 9^4 \equiv 16 \equiv -1 \bmod 17$

Definition ?? (in ??) states that the roots of the μ-th cyclotomic polynomial are the primitive μ-th roots of unity. And Definition ?? (in ??) states that the order of the primitive μ-th roots of unity is μ. These definitions apply to both the cyclotomic polynomials over $X \in \mathbb{C}$ (complex numbers) and the cyclotomic polynomials over $X \in \mathbb{Z}_t$ (ring).

Since $\{2, 8, 15, 9\}$ are the roots of the $(\mu = 8)$-th cyclotomic polynomial $X^4 + 1$ over the ring $\mathbb{Z}_{17}$, they are also the $(\mu = 8)$-th primitive roots of unity of $\mathbb{Z}_{17}$. Therefore, their order (??) is $\mu = 8$ as demonstrated as follows:

$2^8 \equiv 8^8 \equiv 15^8 \equiv 9^8 \equiv 1 \bmod 17$

$2^4 \equiv 8^4 \equiv 15^4 \equiv 9^4 \equiv 16 \not\equiv 1 \bmod 17$

Definition ?? (in ??) and Theorem ?? (??) also state that for each primitive μ-th root of unity ω, $\{\omega^k\}_{\mathsf{gcd}(k,\mu)=1}$ generates all roots of the μ-th cyclotomic polynomial. Notice that in the case of the $(\mu = 8)$-th cyclotomic polynomial $X^4 + 1$, its roots $\{2, 8, 15, 9\}$ generate all $(\mu = 8)$-th roots of unity as follows:

$\{2^1, 2^3, 2^5, 2^7\} \equiv \{8^1, 8^3, 8^5, 8^7\} \equiv \{15^1, 15^3, 15^5, 15^7\} \equiv \{9^1, 9^3, 9^5, 9^7\} \equiv \{2, 8, 15, 9\} \bmod 17$

Among $\{2, 8, 15, 9\}$ as the roots of $X^4 + 1$, let's choose $\omega = 9$ as the base root to construct the encoding matrix $\tilde{W}$ and the decoding matrix $\tilde{W}^*$ as follows:

$$\tilde{W} = \begin{bmatrix} 1 & 1 & 1 & 1 \\ \omega^{J(1)} & \omega^{J(0)} & \omega^{J_*(1)} & \omega^{J_*(0)} \\ (\omega^{J(1)})^2 & (\omega^{J(0)})^2 & (\omega^{J_*(1)})^2 & (\omega^{J_*(0)})^2 \\ (\omega^{J(1)})^3 & (\omega^{J(0)})^3 & (\omega^{J_*(1)})^3 & (\omega^{J_*(0)})^3 \end{bmatrix} \quad \# \text{ where } J(h) = 5^h \bmod 8$$

$$= \begin{bmatrix} 1 & 1 & 1 & 1 \\ 9^5 & 9^1 & 9^3 & 9^7 \\ (9^5)^2 & (9^1)^2 & (9^3)^2 & (9^7)^2 \\ (9^5)^3 & (9^1)^3 & (9^3)^3 & (9^7)^3 \end{bmatrix} \equiv \begin{bmatrix} 1 & 1 & 1 & 1 \\ 8 & 9 & 15 & 2 \\ 13 & 13 & 4 & 4 \\ 2 & 15 & 9 & 8 \end{bmatrix} \bmod 17$$

$$\tilde{W}^* = \begin{bmatrix} 1 & \omega^{J(0)} & (\omega^{J(0)})^2 & (\omega^{J(0)})^3 \\ 1 & \omega^{J(1)} & (\omega^{J(1)})^2 & (\omega^{J(1)})^3 \\ 1 & \omega^{J_*(0)} & (\omega^{J_*(0)})^2 & (\omega^{J_*(0)})^3 \\ 1 & \omega^{J_*(1)} & (\omega^{J_*(1)})^2 & (\omega^{J_*(1)})^3 \end{bmatrix} \equiv \begin{bmatrix} 1 & 9 & 13 & 15 \\ 1 & 8 & 13 & 2 \\ 1 & 2 & 4 & 8 \\ 1 & 15 & 4 & 9 \end{bmatrix} \bmod 17$$

Notice that Theorem ?? (in ??) is demonstrated as follows:

$$\tilde{W}^* \cdot \tilde{W} = \begin{bmatrix} 1 & 9 & 13 & 15 \\ 1 & 8 & 13 & 2 \\ 1 & 2 & 4 & 8 \\ 1 & 15 & 4 & 9 \end{bmatrix} \cdot \begin{bmatrix} 1 & 1 & 1 & 1 \\ 8 & 9 & 15 & 2 \\ 13 & 13 & 4 & 4 \\ 2 & 15 & 9 & 8 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 & 4 \\ 0 & 0 & 4 & 0 \\ 0 & 4 & 0 & 0 \\ 4 & 0 & 0 & 0 \end{bmatrix} = nI_n^R \pmod{17}$$

Now suppose that we encode the following two input vectors (i.e., input vector slots) in $\mathbb{Z}_{17}$:

$\vec{v}_1 = (10, 3, 5, 13)$

$\vec{v}_2 = (2, 4, 3, 6)$

$\vec{v}_1 + \vec{v}_2 = (10, 3, 5, 13) + (2, 4, 3, 6) \equiv (12, 7, 8, 2) \bmod 17$

These two vectors are encoded as follows:

$$\vec{m}_1 = n^{-1}\tilde{W} \cdot I_n^R \cdot \vec{v} = 13 \cdot \begin{bmatrix} 1 & 1 & 1 & 1 \\ 8 & 9 & 15 & 2 \\ 13 & 13 & 4 & 4 \\ 2 & 15 & 9 & 8 \end{bmatrix} \cdot \begin{bmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix} \cdot \begin{bmatrix} 10 \\ 3 \\ 5 \\ 13 \end{bmatrix} \equiv (12, 11, 12, 1) \bmod 17$$

$$\vec{m}_2 = n^{-1}\tilde{W} \cdot I_n^R \cdot \vec{v} = 13 \cdot \begin{bmatrix} 1 & 1 & 1 & 1 \\ 8 & 9 & 15 & 2 \\ 13 & 13 & 4 & 4 \\ 2 & 15 & 9 & 8 \end{bmatrix} \cdot \begin{bmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix} \cdot \begin{bmatrix} 2 \\ 4 \\ 3 \\ 6 \end{bmatrix} \equiv (8, 5, 14, 6) \bmod 17$$

$\Delta M_1(X) = 2 \cdot (12 + 11X + 12X^2 + 1X^3) = 24 + 22X + 24X^2 + 2X^3 \pmod q$ # where $q = 64$
$\Delta M_2(X) = 2 \cdot (8 + 5X + 14X^2 + 6X^3) = 16 + 10X + 28X^2 + 12X^3 \pmod q$
$\Delta M_{1+2}(X) = \Delta M_1(X) + \Delta M_2(X) = \Delta(M_1(X) + M_2(X)) = 40 + 32X + 52X^2 + 14X^3 \pmod q$

Note that the coefficients of the scaled polynomials $\Delta M_1(X)$ and $\Delta M_2(X)$ are still within the range of the ciphertext domain $q = 64$ (which must hold throughout all homomorphic computations to preserve correctness).

We decode $M_1(X)$ and $M_2(X)$ as follows:

$$\vec{m}_1 = \frac{\Delta\vec{m}_1}{\Delta} = \frac{(24, 22, 24, 2)}{2} = (12, 11, 12, 1)$$
$$\vec{m}_2 = \frac{\Delta\vec{m}_2}{\Delta} = \frac{(16, 10, 28, 12)}{2} = (8, 5, 14, 6)$$
$$\vec{m}_{1+2} = \frac{\Delta\vec{m}_{1+2}}{\Delta} = \frac{(40, 32, 52, 14)}{2} = (20, 16, 26, 7)$$

$$\vec{v}_1 = \tilde{W}^* \cdot \vec{m}_1 = \begin{bmatrix} 1 & 9 & 13 & 15 \\ 1 & 8 & 13 & 2 \\ 1 & 2 & 4 & 8 \\ 1 & 15 & 4 & 9 \end{bmatrix} \cdot \begin{bmatrix} 12 \\ 11 \\ 12 \\ 1 \end{bmatrix} = (10, 3, 5, 13) \pmod{17}$$

$$\vec{v}_2 = \tilde{W}^* \cdot \vec{m}_2 = \begin{bmatrix} 1 & 9 & 13 & 15 \\ 1 & 8 & 13 & 2 \\ 1 & 2 & 4 & 8 \\ 1 & 15 & 4 & 9 \end{bmatrix} \cdot \begin{bmatrix} 8 \\ 5 \\ 14 \\ 6 \end{bmatrix} = (2, 4, 3, 6) \pmod{17}$$

$$\vec{v}_{1+2} = \tilde{W}^* \cdot \vec{m}_{1+2} = \begin{bmatrix} 1 & 9 & 13 & 15 \\ 1 & 8 & 13 & 2 \\ 1 & 2 & 4 & 8 \\ 1 & 15 & 4 & 9 \end{bmatrix} \cdot \begin{bmatrix} 20 \\ 16 \\ 26 \\ 7 \end{bmatrix} = (12, 7, 8, 2) \pmod{17}$$

The decoded $\vec{v}_1, \vec{v}_2$, and $\vec{v}_{1+2}$ match the expected values.

D-2.9.5 Rotation Example

Suppose we have the following setup:

$\mu = 16, n = \frac{\mu}{2} = 8,\ t = 17,\ q = 2^6 = 64,\ n^{-1} = 13,\ \Delta = 2$

$\mathcal{R}_{\langle 8,17\rangle} = \mathbb{Z}_{17}[X]/(X^8 + 1)$

The roots of $X^8 + 1 \pmod{17}$ are $X = \{3, 5, 6, 7, 10, 11, 12, 14\}$, as demonstrated as follows:

$3^8 \equiv 5^8 \equiv 6^8 \equiv 7^8 \equiv 10^8 \equiv 11^8 \equiv 12^8 \equiv 14^8 \equiv 16 \equiv -1 \bmod 17$

Among $\{3, 5, 6, 7, 10, 11, 12, 14\}$ as the roots of $X^8 + 1$, let's choose $\omega = 3$ as the base root to construct the encoding matrix $\tilde{W}$ and the decoding matrix $\tilde{W}^*$ as follows:

$$\tilde{W} = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ \omega^{J(3)} & \omega^{J(2)} & \omega^{J(1)} & \omega^{J(0)} & \omega^{J_*(3)} & \omega^{J_*(2)} & \omega^{J_*(1)} & \omega^{J_*(0)} \\ (\omega^{J(3)})^2 & (\omega^{J(2)})^2 & (\omega^{J(1)})^2 & (\omega^{J(0)})^2 & (\omega^{J_*(3)})^2 & (\omega^{J_*(2)})^2 & (\omega^{J_*(1)})^2 & (\omega^{J_*(0)})^2 \\ (\omega^{J(3)})^3 & (\omega^{J(2)})^3 & (\omega^{J(1)})^3 & (\omega^{J(0)})^3 & (\omega^{J_*(3)})^3 & (\omega^{J_*(2)})^3 & (\omega^{J_*(1)})^3 & (\omega^{J_*(0)})^3 \\ (\omega^{J(3)})^4 & (\omega^{J(2)})^4 & (\omega^{J(1)})^4 & (\omega^{J(0)})^4 & (\omega^{J_*(3)})^4 & (\omega^{J_*(2)})^4 & (\omega^{J_*(1)})^4 & (\omega^{J_*(0)})^4 \\ (\omega^{J(3)})^5 & (\omega^{J(2)})^5 & (\omega^{J(1)})^5 & (\omega^{J(0)})^5 & (\omega^{J_*(3)})^5 & (\omega^{J_*(2)})^5 & (\omega^{J_*(1)})^5 & (\omega^{J_*(0)})^5 \\ (\omega^{J(3)})^6 & (\omega^{J(2)})^6 & (\omega^{J(1)})^6 & (\omega^{J(0)})^6 & (\omega^{J_*(3)})^6 & (\omega^{J_*(2)})^6 & (\omega^{J_*(1)})^6 & (\omega^{J_*(0)})^6 \\ (\omega^{J(3)})^7 & (\omega^{J(2)})^7 & (\omega^{J(1)})^7 & (\omega^{J(0)})^7 & (\omega^{J_*(3)})^7 & (\omega^{J_*(2)})^7 & (\omega^{J_*(1)})^7 & (\omega^{J_*(0)})^7 \end{bmatrix}$$

$$\equiv \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 12 & 14 & 5 & 3 & 10 & 11 & 7 & 6 \\ 8 & 9 & 8 & 9 & 15 & 2 & 15 & 2 \\ 11 & 7 & 6 & 10 & 14 & 5 & 3 & 12 \\ 13 & 13 & 13 & 13 & 4 & 4 & 4 & 4 \\ 3 & 12 & 14 & 5 & 6 & 10 & 11 & 7 \\ 2 & 15 & 2 & 15 & 9 & 8 & 9 & 8 \\ 7 & 6 & 10 & 11 & 5 & 3 & 12 & 14 \end{bmatrix} \bmod 17$$

$$\tilde{W}^* = \begin{bmatrix} 1 & \omega^{J(0)} & (\omega^{J(0)})^2 & (\omega^{J(0)})^3 & (\omega^{J(0)})^4 & (\omega^{J(0)})^5 & (\omega^{J(0)})^6 & (\omega^{J(0)})^7 \\ 1 & \omega^{J(1)} & (\omega^{J(1)})^2 & (\omega^{J(1)})^3 & (\omega^{J(1)})^4 & (\omega^{J(1)})^5 & (\omega^{J(1)})^6 & (\omega^{J(1)})^7 \\ 1 & \omega^{J(2)} & (\omega^{J(2)})^2 & (\omega^{J(2)})^3 & (\omega^{J(2)})^4 & (\omega^{J(2)})^5 & (\omega^{J(2)})^6 & (\omega^{J(2)})^7 \\ 1 & \omega^{J(3)} & (\omega^{J(3)})^2 & (\omega^{J(3)})^3 & (\omega^{J(3)})^4 & (\omega^{J(3)})^5 & (\omega^{J(3)})^6 & (\omega^{J(3)})^7 \\ 1 & \omega^{J_*(0)} & (\omega^{J_*(0)})^2 & (\omega^{J_*(0)})^3 & (\omega^{J_*(0)})^4 & (\omega^{J_*(0)})^5 & (\omega^{J_*(0)})^6 & (\omega^{J_*(0)})^7 \\ 1 & \omega^{J_*(1)} & (\omega^{J_*(1)})^2 & (\omega^{J_*(1)})^3 & (\omega^{J_*(1)})^4 & (\omega^{J_*(1)})^5 & (\omega^{J_*(1)})^6 & (\omega^{J_*(1)})^7 \\ 1 & \omega^{J_*(2)} & (\omega^{J_*(2)})^2 & (\omega^{J_*(2)})^3 & (\omega^{J_*(2)})^4 & (\omega^{J_*(2)})^5 & (\omega^{J_*(2)})^6 & (\omega^{J_*(2)})^7 \\ 1 & \omega^{J_*(3)} & (\omega^{J_*(3)})^2 & (\omega^{J_*(3)})^3 & (\omega^{J_*(3)})^4 & (\omega^{J_*(3)})^5 & (\omega^{J_*(3)})^6 & (\omega^{J_*(3)})^7 \end{bmatrix}$$

$$\equiv \begin{bmatrix} 1 & 3 & 9 & 10 & 13 & 5 & 15 & 11 \\ 1 & 5 & 8 & 6 & 13 & 14 & 2 & 10 \\ 1 & 14 & 9 & 7 & 13 & 12 & 15 & 6 \\ 1 & 12 & 8 & 11 & 13 & 3 & 2 & 7 \\ 1 & 6 & 2 & 12 & 4 & 7 & 8 & 14 \\ 1 & 7 & 15 & 3 & 4 & 11 & 9 & 12 \\ 1 & 11 & 2 & 5 & 4 & 10 & 8 & 3 \\ 1 & 10 & 15 & 14 & 4 & 6 & 9 & 5 \end{bmatrix} \bmod 17$$

Notice that Theorem **??** (in **??**) is demonstrated as follows:

$$\tilde{W}^* \cdot \tilde{W} = \begin{bmatrix} 1 & 3 & 9 & 10 & 13 & 5 & 15 & 11 \\ 1 & 5 & 8 & 6 & 13 & 14 & 2 & 10 \\ 1 & 14 & 9 & 7 & 13 & 12 & 15 & 6 \\ 1 & 12 & 8 & 11 & 13 & 3 & 2 & 7 \\ 1 & 6 & 2 & 12 & 4 & 7 & 8 & 14 \\ 1 & 7 & 15 & 3 & 4 & 11 & 9 & 12 \\ 1 & 11 & 2 & 5 & 4 & 10 & 8 & 3 \\ 1 & 10 & 15 & 14 & 4 & 6 & 9 & 5 \end{bmatrix} \cdot \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 12 & 14 & 5 & 3 & 10 & 11 & 7 & 6 \\ 8 & 9 & 8 & 9 & 15 & 2 & 15 & 2 \\ 11 & 7 & 6 & 10 & 14 & 5 & 3 & 12 \\ 13 & 13 & 13 & 13 & 4 & 4 & 4 & 4 \\ 3 & 12 & 14 & 5 & 6 & 10 & 11 & 7 \\ 2 & 15 & 2 & 15 & 9 & 8 & 9 & 8 \\ 7 & 6 & 10 & 11 & 5 & 3 & 12 & 14 \end{bmatrix}$$

$$= \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 8 \\ 0 & 0 & 0 & 0 & 0 & 0 & 8 & 0 \\ 0 & 0 & 0 & 0 & 0 & 8 & 0 & 0 \\ 0 & 0 & 0 & 0 & 8 & 0 & 0 & 0 \\ 0 & 0 & 0 & 8 & 0 & 0 & 0 & 0 \\ 0 & 0 & 8 & 0 & 0 & 0 & 0 & 0 \\ 0 & 8 & 0 & 0 & 0 & 0 & 0 & 0 \\ 8 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} = nI_n^R \pmod{17}$$

Now suppose that we encode the following input vector (i.e., input vector slots) in $\mathbb{Z}_{17}$:
$\vec{v} = (1, 2, 3, 4, 5, 6, 7, 8)$
By rotating this vector by 3 positions to the left (i.e., the first-half slots and the second-half slots separately wrapping around within their own group), we get a new vector:
$\vec{v}_r = (4, 1, 2, 3, 8, 5, 6, 7) \bmod 17$

$\vec{v}$ is encoded as follows:

$$\vec{m} = n^{-1}\tilde{W} \cdot I_n^R \cdot \vec{v}$$

$$= 13 \cdot \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 12 & 14 & 5 & 3 & 10 & 11 & 7 & 6 \\ 8 & 9 & 8 & 9 & 15 & 2 & 15 & 2 \\ 11 & 7 & 6 & 10 & 14 & 5 & 3 & 12 \\ 13 & 13 & 13 & 13 & 4 & 4 & 4 & 4 \\ 3 & 12 & 14 & 5 & 6 & 10 & 11 & 7 \\ 2 & 15 & 2 & 15 & 9 & 8 & 9 & 8 \\ 7 & 6 & 10 & 11 & 5 & 3 & 12 & 14 \end{bmatrix} \cdot \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \cdot \begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \\ 8 \end{bmatrix}$$

$\equiv (13, 16, 10, 5, 9, 12, 7, 1) \bmod 17$

$\Delta M(X) = 2 \cdot (13 + 16X + 10X^2 + 5X^3 + 9X^4 + 12X^5 + 7X^6 + X^7)$
$= 26 + 32X + 20X^2 + 10X^3 + 18X^4 + 24X^5 + 14X^6 + 2X^7 \pmod{q}$ # where $q = 64$

$\Delta M(X^{J(3)}) = \Delta M(X^{13}) = 26 + 24X - 20X^2 - 2X^3 + 18X^4 - 32X^5 - 14X^6 + 10X^7 \pmod{q}$

Note that the coefficients of the scaled polynomials $\Delta M(X^{J(3)})$ are still within the range of the ciphertext domain $q = 64$ (which must hold throughout all homomorphic computations to preserve correctness).
We decode $M(X^{J(3)})$ as follows:

$$\vec{m}_{J(3)} = \frac{\Delta\vec{m}_{J(3)}}{\Delta} = \frac{(26, 24, -20, -2, 18, -32, -14, 10)}{2} = (13, 12, -10, -2, 18, -32, -14, 10) \pmod{17}$$

$$\vec{v}_{J(3)} = \tilde{W}^* \cdot \vec{m}_{J(3)} = \begin{bmatrix} 1 & 3 & 9 & 10 & 13 & 5 & 15 & 11 \\ 1 & 5 & 8 & 6 & 13 & 14 & 2 & 10 \\ 1 & 14 & 9 & 7 & 13 & 12 & 15 & 6 \\ 1 & 12 & 8 & 11 & 13 & 3 & 2 & 7 \\ 1 & 6 & 2 & 12 & 4 & 7 & 8 & 14 \\ 1 & 7 & 15 & 3 & 4 & 11 & 9 & 12 \\ 1 & 11 & 2 & 5 & 4 & 10 & 8 & 3 \\ 1 & 10 & 15 & 14 & 4 & 6 & 9 & 5 \end{bmatrix} \cdot \begin{bmatrix} 13 \\ 12 \\ -10 \\ -2 \\ 18 \\ -32 \\ -14 \\ 10 \end{bmatrix}$$

$$= (4, 1, 2, 3, 8, 5, 6, 7) \pmod{17}$$

$$= \vec{v}_r$$

The decoded $\vec{v}_{J(3)}$ matches the expected rotated input vector $\vec{v}_r$.

In practice, we do not directly update $\Delta M(X)$ to $\Delta M(X^{J(3)})$, because we would not have access to the plaintext polynomial $M(X)$ unless we have the secret key $S(X)$. Therefore, we instead update $\mathsf{ct} = (A(X), B(X))$ to $\mathsf{ct}^{\langle h=3\rangle} = (A(X^{J(3)}), B(X^{J(3)}))$, which is equivalent to homomorphically rotating the encrypted input vector slots. Then, decrypting $\mathsf{ct}^{\langle h=3\rangle}$ and decoding it would output $\vec{v}_r$.

Source Code: Examples of BFV's batch encoding and homomorphic input vector rotation can be executed by running this Python script.

D-2.10 Application: Matrix Multiplication

BFV has no clean way to do a homomorphic dot product between two vectors (i.e., $\vec{v}_1 \cdot \vec{v}_2$), because the last step of a vector dot product requires summation of all slot-wise intermediate values (i.e., $v_{1,1}v_{2,1} + v_{1,2}v_{2,2} + \cdots + v_{1,n}v_{2,n}$). However, each slot in BFV's batch encoding is independent from each other, which cannot be simply added up across slots (i.e., input vector elements). Instead, we need n copies of the multiplied ciphertexts and properly align their slots by many rotation operations before adding them up. Meanwhile, the homomorphic input vector slot rotation scheme can be effectively used when we homomorphically multiply a plaintext matrix with an encrypted vector. Remember that given a matrix A and vector $\vec{x}$ (Definition ?? in ??):

$$A = \begin{bmatrix} a_{\langle 0,0\rangle} & a_{\langle 0,1\rangle} & a_{\langle 0,2\rangle} & \cdots & a_{\langle 0,n-1\rangle} \\ a_{\langle 1,0\rangle} & a_{\langle 1,1\rangle} & a_{\langle 1,2\rangle} & \cdots & a_{\langle 1,n-1\rangle} \\ a_{\langle 2,0\rangle} & a_{\langle 2,1\rangle} & a_{\langle 2,2\rangle} & \cdots & a_{\langle 2,n-1\rangle} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ a_{\langle m-1,0\rangle} & a_{\langle m-1,1\rangle} & a_{\langle m-1,2\rangle} & \cdots & a_{\langle m-1,n-1\rangle} \end{bmatrix} = \begin{bmatrix} \vec{a}_{\langle 0,*\rangle} \\ \vec{a}_{\langle 1,*\rangle} \\ \vec{a}_{\langle 2,*\rangle} \\ \vdots \\ \vec{a}_{\langle m-1,*\rangle} \end{bmatrix}, \quad \vec{x} = (x_0, x_1, \cdots, x_{n-1})$$

The result of $A \cdot \vec{x}$ is an m-dimensional vector computed as:

$$A \cdot \vec{x} = \left(\vec{a}_{\langle 0,*\rangle} \cdot \vec{x},\ \vec{a}_{\langle 1,*\rangle} \cdot \vec{x},\ \cdots,\ \vec{a}_{\langle m-1,*\rangle} \cdot \vec{x}\right) = \left(\sum_{i=0}^{n-1} a_{0,i} \cdot x_i,\ \sum_{i=0}^{n-1} a_{1,i} \cdot x_i,\ \cdots,\ \sum_{i=0}^{n-1} a_{m-1,i} \cdot x_i\right)$$

Let's define $\rho(\vec{v}, h)$ as the rotation of $\vec{v}$ by h positions to the left. And remember that the Hadamard dot product (Definition ?? in ??) is defined as slot-wise multiplication of two vectors:

$\vec{a} \odot \vec{b} = (a_0 b_0,\ a_1 b_1,\ \cdots,\ a_{n-1} b_{n-1})$

Let's define n distinct diagonal vector $\vec{u}_i$ extracted from matrix A as follows:
$\vec{u}_i = \{a_{\langle j \bmod m,\ i+j \bmod n \rangle}\}_{j=0}^{n-1}$

Then, the original matrix-to-vector multiplication formula can be equivalently constructed as follows:
$A \cdot \vec{x} = \vec{u}_0 \odot \rho(\vec{x}, 0) \ + \ \vec{u}_1 \odot \rho(\vec{x}, 1) \ + \ \cdots \ + \ \vec{u}_{n-1} \odot \rho(\vec{x}, n-1)$
, whose computation result is equivalent to $A \cdot \vec{x}$. The above formula is compatible with homomorphic computation, because BFV supports Hadamard dot product between input vectors as a ciphertext-to-plaintext multiplication between their polynomial-encoded forms (**??**), and BFV also supports $\rho(\vec{v}, h)$ as homomorphic input vector slot rotation (**??**). After homomorphically computing the above formula, we can consider only the first m (out of n) resulting input vector slots to store the result of $A \cdot \vec{x}$.

D-2.11 Noise Bootstrapping

- **Reference 1:** Bootstrapping for BGV and BFV Revisited [**?**]
- **Reference 2:** Bootstrapping for HELib [**?**]
- **Reference 3:** A Note on Lower Digits Extraction Polynomial for Bootstrapping [**?**]
- **Reference 4:** Fully Homomorphic Encryption for Cyclotomic Prime Moduli [**?**]

In BFV, continuous ciphertext-to-ciphertext multiplication increases the noise in a multiplicative manner, and once the noise overflows the message bits, then the message gets corrupted. Bootstrapping is a process of resetting the grown noise.

D-2.11.1 High-level Idea

In this subsection, we will assume the plaintext modulus $t = p$, a prime number. Although t can be generalized as $t = p^r$ where $r \in \mathbb{I}$ and $r \geq 1$ (Summary **??** in **??**), we will explain BFV's bootstrapping assuming $t = p$ for simplicity, and then generalize t as $t = p^r$ in the end.

The core idea of the BFV bootstrapping is to homomorphically evaluate a special polynomial $G_\varepsilon(x)$, a digit extraction polynomial modulo p^ε (for some positive integer ε), where the input to $G_\varepsilon(x)$ is a noisy plaintext value modulo p^ε and the output is a noise-free plaintext value modulo p^ε, where the noise located at the least significant digits in a base-p (prime) representation gets zeroed out. For example, $G_\varepsilon(3p^3 + 4p^2 + 6p + 2) = 3p^3 + 4p^2 + 6p \bmod p^\varepsilon$. Given the noise resides in the least significant $\varepsilon - 1$ digits in base-p representation, we can homomorphically recursively evaluate $G_\varepsilon(x)$ total $\varepsilon - 1$ times in a row, which zeros out the least significant (base-p) $\varepsilon - 1$ digits of input x. To homomorphically evaluate the noisy plaintext through $G_\varepsilon(x) \bmod p^\varepsilon$, we need to first switch the plaintext modulus from t to p^ε, where $q \gg p^\varepsilon > p = t$. The larger ε is, the more likely that the noise gets successfully zeroed out, but instead the computation overhead becomes larger. If ε is small, the computation gets faster, but the digit-wise distance between the noise and the plaintext gets smaller, potentially corrupting the plaintext during bootstrapping, because removing the most significant noise digit may remove the least significant plaintext digit as well. Therefore, ε should be chosen carefully.

The technical details of the BFV bootstrapping procedure are as follows. Suppose we have an RLWE ciphertext $(A, B) = \mathsf{RLWE}_{S,\sigma}(\Delta M) \bmod q$, where $A \cdot S + B = \Delta M + E$, $\Delta = \left\lfloor \frac{q}{t} \right\rfloor$ and $t = p$ (i.e., the plaintext modulus is a prime).

1. **Modulus Switch ($q \to p^\varepsilon$):** Scale down the ciphertext modulus from $(A, B) \bmod q$ to $\left(\left\lceil \frac{p^\varepsilon}{q} \cdot A \right\rfloor, \left\lceil \frac{p^\varepsilon}{q} \cdot B \right\rfloor \right) = (A', B') \bmod p^\varepsilon$, where $p^\varepsilon \ll q$. The purpose of this modulus switch is to change the plaintext modulus to p^ε, which is required to use the digit extraction polynomial $G_\varepsilon(x)$ (because we need to represent the input to $G_\varepsilon(x)$ as a base-p number in order to interpret it as a modp^ε value). Notice that $\mathsf{RLWE}^{-1}_{S,\sigma}(\mathsf{ct} = (A', B')) = p^{\varepsilon-1}M + E'$, where $E' \approx \frac{p^\varepsilon}{q} \cdot E + \left(\left\lfloor \frac{q}{p} \right\rfloor \cdot \frac{p^\varepsilon}{q} - p^{\varepsilon-1} \right) \cdot M$ # the modulus switch error of $E \to E'$ plus the modulus switch error of the plaintext's scaling factor $\left\lfloor \frac{q}{t} \right\rfloor \to p^{\varepsilon-1}$

2. **Homomorphic Decryption:** Suppose we have the *bootstrapping key* $\mathsf{RLev}^{\beta,l}_{S,\sigma}(S) \bmod q$, which is the secret key S encrypted by S (itself) modulo q. Using this *encrypted* secret key, we *homomorphically* decrypt (A', B') as follows:
$A' \cdot \mathsf{RLWE}_{S,\sigma}(\Delta' S) + B'$ # where $\mathsf{RLWE}_{S,\sigma}(\Delta' S)$ is a modulo-q ciphertext that encrypts a modulo-p^ε plaintext S whose scaling factor $\Delta' = \frac{q}{p^\varepsilon}$

$= \mathsf{RLWE}_{S,\sigma}(\Delta'(A' \cdot S)) + B' \bmod q$
$= \mathsf{RLWE}_{S,\sigma}(\Delta' \cdot (A' \cdot S + B')) \bmod q$
$= \mathsf{RLWE}_{S,\sigma}(\Delta' \cdot (p^{\varepsilon-1}M + E' + Kp^\varepsilon)) \bmod q$ # K is some integer polynomials to represent the coefficient values that wrap around p^ε

Let's see how we derived the above relation. Suppose we compute $A' \cdot S + B' \bmod p^\varepsilon$, whose output will be $p^{\varepsilon-1}M + E'$. Now, instead of using the plaintext secret key S, we use an encrypted secret key $\mathsf{RLWE}_{S,\sigma}(\Delta' S)$, where S is a plaintext modulo p^ε, its scaling factor $\Delta' = \left\lfloor \frac{q}{p^\varepsilon} \right\rfloor$, and the ciphertext encrypting S is in modulo q. Then, the result of homomorphically computing $A' \cdot \mathsf{RLWE}_{S,\sigma}(\Delta' S) + B'$ will be an encryption of $p^{\varepsilon-1}M + E' + Kp^\varepsilon$, where Kp^ε stands for the wrapping-around coefficient values of the multiples of p^ε.
Notice that we did not reduce Kp^ε by modulo p^ε during homomorphic decryption (without modulo-q reduction), because such a homomorphic modulo reduction is not directly doable. Instead, we will handle Kp^ε in the later digit extraction step.
For simplicity, we will denote $Z = p^{\varepsilon-1}M + E' \bmod p^\varepsilon$.

3. **CoeffToSlot:** Move the (encrypted) polynomial Z's coefficients $z_0, z_i, \cdots, z_{n-1}$ to the input vector slots of an RLWE ciphertext. This is done by computing:
$\mathsf{RLWE}_{S,\sigma}(Z) \cdot n^{-1} \cdot \tilde{W} \cdot I^n_R$
, where $n^{-1} \cdot \tilde{W} \cdot I^n_R$ is the batch encoding matrix that converts input vector slot values into polynomial coefficients (Summary **??** in **??**).

4. **Digit Extraction:** We design a polynomial $G_\varepsilon(x)$ (a digit extraction polynomial) zeros out the least significant base-p digit(s) by recursively evaluating the polynomial. Using each z_i (i.e., the i-th coefficient of Z) as an input, we recursively evaluate $G_\varepsilon(z_i)$ total $\varepsilon - 1$ times consecutively,

which effectively zeros out the least significant $\varepsilon - 1$ digits of the base-p representation of z_i (i.e., noise value), and keeps only the most significant base-p digit of x (i.e., plaintext value). Throughout the digit extraction, the value stored at the input vector slots of the ciphertext gets updated from $p^{\varepsilon-1}M + E' + Kp^{\varepsilon}$ to $p^{\varepsilon-1}M + K'p^{\varepsilon}$.

5. **SlotToCoeff:** Homomorphically move each input vector slot's value back to the (encrypted) polynomial coefficient position. This is done by homomorphically multiplying the decoding matrix $\tilde{W}^*$ to the ciphertext (Summary **??** in **??**).

6. **Scaling Factor Re-interpretation:** At this point, we have the ciphertext $\mathsf{RLWE}_{S,\sigma}(\Delta' \cdot (p^{\varepsilon-1}M + K'p^{\varepsilon}))$ mod q, where the plaintext modulus is p^{ε} and the plaintext scaling factor $\Delta' = \frac{q}{p^{\varepsilon}}$. Without doing any actual additional computation, we can view this ciphertext as $\mathsf{RLWE}_{S,\sigma}(\frac{q}{p}M)) = \mathrm{mod} q$, which is an encryption of plaintext modulus p whose scaling factor $\Delta' = \frac{q}{p}$. This is because:
$\mathsf{RLWE}_{S,\sigma}(\Delta' \cdot (p^{\varepsilon-1}M + K'p^{\varepsilon})) = \mathsf{RLWE}_{S,\sigma}\Big(\frac{q}{p^{\varepsilon}} \cdot (p^{\varepsilon-1}M + K'p^{\varepsilon})\Big)$
$= \mathsf{RLWE}_{S,\sigma}\left(\frac{q}{p}M + K'q\right)$
$= \mathsf{RLWE}_{S,\sigma}\left(\frac{q}{p}M\right)$
In other words, we can view the ciphertext $\mathsf{RLWE}_{S,\sigma}(\Delta' \cdot (p^{\varepsilon-1}M + K'p^{\varepsilon}))$ mod q whose plaintext modulus is p^{ε} and scaling factor $\Delta' = \frac{q}{p^{\varepsilon}}$ as another ciphertext $\mathsf{RLWE}_{S,\sigma}(\Delta M)$ mod q whose plaintext modulus is p and scaling factor $\Delta = \frac{q}{p}$.

Notice that the final ciphertext $\mathsf{RLWE}_{S,\sigma}(\Delta' \cdot M)$'s scaling factor stays the same as before the bootstrapping, while the original noises E and E' have been eliminated. On the other hand, the homomorphic operation of CoeffToSlot, digit extraction, and SlotToCoeff must have accumulated additional noises, but their size is fixed and smaller than E and E'.

Next, we will explain each step more in detail.

D-2.11.2 Modulus Switch

The first step of the BFV bootstrapping is to do a modulus switch from q to some prime power modulus p^{ε} where $p^{\varepsilon} \ll q$. Before the bootstrapping, suppose the encrypted plaintext with noise is $\Delta M + E$ mod q. Then, after the modulus switch from $q \to p^{\varepsilon}$, the plaintext would scale down to $p^{\varepsilon-1}M + E'$ mod p^{ε}, where E' roughly contains $\left\lceil \frac{p^{\varepsilon}}{q}E \right\rfloor$ plus the modulus switching noise of the plaintext's scaling factor $\Delta \to p^{\varepsilon-1}$. The goal of the BFV bootstrapping is to zero out this noise E'.

D-2.11.3 Homomorphic Decryption

Let's denote the modulus-switched noisy plaintext as $Z = p^{\varepsilon-1}M + E'$ mod p^{ε}. We further denote polynomial Z's each degree term's coefficient z_i as base-p number as follows:

$z_i = z_{i,\varepsilon-1}p^{\varepsilon-1} + z_{i,\varepsilon-2}p^{\varepsilon-2} + \cdots + z_{i,1}p + z_{i,0} \bmod p^{\varepsilon}$

Then, $z_i \bmod p^{\varepsilon}$ is a base-p number comprising ε digits: $\{z_{i,\varepsilon-1}, z_{i,\varepsilon-2}, \cdots, z_{i,0}\}$

We assume that the highest base-p digit index for the noise is $\varepsilon - 2$, which is equivalent to the noise budget, and the pure plaintext portion solely resides at the base-p digit index $\varepsilon - 1$ (i.e., the most significant base-p digit in modulo p^{ε}). Given this assumption, we extract the noise-free plaintext by computing the following:

$\left\lceil \frac{z_i}{p^{\varepsilon-1}} \right\rfloor \bmod p = z_{i,\varepsilon-1}$ # where the noise is assumed to be smaller than $\frac{p^{\varepsilon-1}}{2}$

The above formula is equivalent to shifting the base-p number z_i by $\varepsilon - 1$ digits to the right (and rounding the decimal value). However, remember that we don't have direct access to polynomial $Z = p^{\varepsilon-1}M + E' \bmod p^{\varepsilon}$ unless we have the secret key S to decrypt the ciphertext storing the plaintext. Instead, we can only derive Z as an encrypted form. Specifically, we can *homomorphically* decrypt the modulus-switched ciphertext (A', B') by using the *encrypted* secret key $\mathsf{RLWE}_{S,\sigma}(\Delta' S)$ as a *bootstrapping key*. For this ciphertext $\mathsf{RLWE}_{S,\sigma}(\Delta' S)$, the plaintext modulus is p^{ε}, the plaintext scaling factor is $\Delta' = \frac{q}{p^{\varepsilon}}$, and the ciphertext modulus is q. With this encrypted secret key S, we homomorphically decrypt the encrypted Z as follows:

$\left(\left\lceil \frac{p^{\varepsilon}}{q} \cdot A \right\rfloor, \left\lceil \frac{p^{\varepsilon}}{q} \cdot B \right\rfloor\right) = (A', B') \bmod p^{\varepsilon}$

$A' \cdot \mathsf{RLWE}_{S,\sigma}(\Delta' S) + B' \bmod q$
$= \mathsf{RLWE}_{S,\sigma}(\Delta'(A' \cdot S)) + B' \bmod q$
$= \mathsf{RLWE}_{S,\sigma}(\Delta' \cdot (A' \cdot S + B')) \bmod q$
$= \mathsf{RLWE}_{S,\sigma}(\Delta' \cdot (p^{\varepsilon-1}M + E' + Kp^{\varepsilon})) \bmod q$ # K is some integer polynomials to represent the coefficient values that wrap around modulo p^{ε} as multiples of p^{ε}

$= \mathsf{RLWE}_{S,\sigma}(\Delta' Z) \bmod q$

During this homomorphic decryption, we did not reduce the plaintext result by modulo p^{ε}, because the homomorphic decryption is a ciphertext-to-plaintext multiplication and addition done in the ciphertext modulus q (not p^{ε}) by using A' and B' as plaintexts (with the plaintext modulus p^{e}) and $\mathsf{RLWE}_{S,\sigma}(\Delta' S)$ as a ciphertext (with the ciphertext modulus q). This is why the wrapping term Kp^{ε} is preserved in the plaintext after the homomorphic decryption– we will handle this term at the later stage of bootstrapping. Also, notice that the computation of $A' \cdot \mathsf{RLWE}_{S,\sigma}(\Delta' S)$ would not generate much noise. This is because A' is a plaintext modulo p^{ε}, and thus the new noise generated by ciphertext-to-plaintext multiplication is $A' \cdot E_s$ (where E_s is the encryption noise of $\mathsf{RLWE}_{S,\sigma}(\Delta' S)$). Since the ciphertext modulus $q \gg p^{\varepsilon}$, $q \gg A' \cdot E_s$.

Once we have derived $\mathsf{RLWE}_{S,\sigma}(\Delta' Z)$, our next step is to remove the noise in the lower $\varepsilon - 1$ digits (in terms of base-p representation) of each z_i for $0 \leq i \leq n-1$. This is equivalent to transforming noisy $\mathsf{RLWE}_{S,\sigma}(\Delta' Z) = \mathsf{RLWE}_{S,\sigma}(\Delta' \cdot (p^{\varepsilon-1}M + E' + Kp^{\varepsilon}))$ into noise-free $\mathsf{RLWE}_{S,\sigma}(\Delta M))$ where $\Delta = \frac{q}{p}$. BFV's solution to do this is to design a p-degree polynomial function which computes the same logical result as $\left\lceil \frac{z_i}{p^{\varepsilon-1}} \right\rfloor \bmod p$. We will later explain how to design this polynomial by using the digit extraction polynomial $G_{\varepsilon}(x)$ (**??**).

However, in order to *homomorphically* evaluate this polynomial at each coefficient z_i given the

ciphertext $\mathsf{RLWE}_{S,\sigma}(\Delta' Z)$, we need to move polynomial Z's each coefficient z_i to the input vector slots of an RLWE ciphertext. This is because BFV supports homomorphic batched $(+, \cdot)$ operations based on input vector slots of ciphertexts as operands. Therefore, we need to evaluate the noise-removing polynomial $G_\varepsilon(x)$ based on the values stored in the input vector slots of a ciphertext.

In the next sub-section, we will explain the CoeffToSlot procedure, a process of moving polynomial coefficients into input vector slots of a ciphertext *homomorphically*.

D-2.11.4 CoeffToSlot and SlotToCoeff

The goal of the CoeffToSlot step is to homomorphically move polynomial Z's coefficients z_i to input vector slots.

In Summary **??** (in **??**), we learned that the encoding formula for converting a vector of input slots $\vec{v}$ into a vector of polynomial coefficients $\vec{m}$ is: $\vec{m} = n^{-1} \cdot \tilde{W} \cdot I_n^R \cdot \vec{v}$, where $\tilde{W}$ is a basis of the n-dimensional vector space crafted as follows:

$$\tilde{W} = \begin{bmatrix} 1 & 1 & \cdots & 1 & 1 & 1 & \cdots & 1 \\ (\omega^{J(\frac{n}{2}-1)}) & (\omega^{J(\frac{n}{2}-2)}) & \cdots & (\omega^{J(0)}) & (\omega^{J_*(\frac{n}{2}-1)}) & (\omega^{J_*(\frac{n}{2}-2)}) & \cdots & (\omega^{J_*(0)}) \\ (\omega^{J(\frac{n}{2}-1)})^2 & (\omega^{J(\frac{n}{2}-2)})^2 & \cdots & (\omega^{J(0)})^2 & (\omega^{J_*(\frac{n}{2}-1)})^2 & (\omega^{J_*(\frac{n}{2}-2)})^2 & \cdots & (\omega^{J_*(0)})^2 \\ \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \vdots \\ (\omega^{J(\frac{n}{2}-1)})^{n-1} & (\omega^{J(\frac{n}{2}-2)})^{n-1} & \cdots & (\omega^{J(0)})^{n-1} & (\omega^{J_*(\frac{n}{2}-1)})^{n-1} & (\omega^{J_*(\frac{n}{2}-2)})^{n-1} & \cdots & (\omega^{J_*(0)})^{n-1} \end{bmatrix}$$

\# where the rotation helper function $J(h) = 5^h \bmod 2n$

Therefore, given the input ciphertext $\mathsf{ct} = \mathsf{RLWE}_{S,\sigma}(\Delta' Z) \bmod q$, we can understand its input vector slots as storing some values such that multiplying $n^{-1} \cdot \tilde{W} \cdot I_n^R$ to each of them turns them into a coefficient z_i of polynomial Z. This implies that if we *homomorphically* multiply $n^{-1} \cdot \tilde{W} \cdot I_n^R$ to the input vector slots of $\mathsf{RLWE}_{S,\sigma}(\Delta' Z)$, then the resulting ciphertext's n-dimensional input vector slots will contain the n coefficients of Z, which is equivalent to moving the coefficients of Z to the input vector slots. Therefore, the CoeffToSlot step is equivalent to homomorphically computing $n^{-1} \cdot \tilde{W} \cdot I_n^R \cdot \mathsf{RLWE}_{S,\sigma}(\Delta' Z)$. We can homomorphically compute matrix-vector multiplication by using the technique explained in **??**.

After the CoeffToSlot step, we can homomorphically eliminate the noise in the lower $\varepsilon - 1$ base-p digits of each z_i by homomorphically evaluating the noise-removing polynomial (to be explained in the next subsection).

After we get noise-free coefficients of Z, we need to move them back from the input vector slots to their original coefficient positions. This step is called SlotToCoeff, which is an exact inverse procedure of CoeffToSlot. We also learned in Summary **??** (in **??**) that the inverse matrix of $n^{-1} \cdot \tilde{W} \cdot I_n^R$ is $\tilde{W}^*$, where:

$$\tilde{W}^* = \begin{bmatrix} 1 & (\omega^{J(0)}) & (\omega^{J(0)})^2 & \cdots & (\omega^{J(0)})^{n-1} \\ 1 & (\omega^{J(1)}) & (\omega^{J(1)})^2 & \cdots & (\omega^{J(1)})^{n-1} \\ 1 & (\omega^{J(2)}) & (\omega^{J(2)})^2 & \cdots & (\omega^{J(2)})^{n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & (\omega^{J(\frac{n}{2}-1)}) & (\omega^{J(\frac{n}{2}-1)})^2 & \cdots & (\omega^{J(\frac{n}{2}-1)})^{n-1} \\ 1 & (\omega^{J_*(0)}) & (\omega^{J_*(0)})^2 & \cdots & (\omega^{J_*(0)})^{n-1} \\ 1 & (\omega^{J_*(1)}) & (\omega^{J_*(1)})^2 & \cdots & (\omega^{J_*(1)})^{n-1} \\ 1 & (\omega^{J_*(2)}) & (\omega^{J_*(2)})^2 & \cdots & (\omega^{J_*(2)})^{n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & (\omega^{J_*(\frac{n}{2}-1)}) & (\omega^{J_*(\frac{n}{2}-1)})^2 & \cdots & (\omega^{J_*(\frac{n}{2}-1)})^{n-1} \end{bmatrix}$$

Therefore, the SlotToCoeff step is equivalent to homomorphically multiplying $\tilde{W}^*$ to the output of the noise-eliminating polynomial evaluation.

In the next subsection, we will learn how to design the core algorithm of BFV, the noise elimination polynomial, based on the digit extraction polynomial $G_\varepsilon(x)$.

D-2.11.5 Digit Extraction

Remember we defined polynomial Z as the scaled noisy plaintext: $Z = p^{\varepsilon-1}M + E' + Kp^\varepsilon = p^{\varepsilon-1}M + E' \bmod p^\varepsilon$, and each z_i is the i-th coefficient of Z (where $0 \le i \le n-1$). The goal of the digit extraction step is to homomorphically zero out the lower (base-p) $\varepsilon - 1$ digits of each z_i, where the noise resides.

First, we always think of z_i as a base-p number (since this is a modulo-p^ε value) as follows:
$z_i = z_{i,\varepsilon-1}p^{\varepsilon-1} + z_{i,\varepsilon-2}p^{\varepsilon-2} + \cdots + z_{i,2}p^2 + z_{i,1}p + z_{i,0} \bmod p^\varepsilon$

Next, we define a new notation that denotes z_i in a different way as follows:
$z_i = d_0 + \left(\sum_{j=\varepsilon'}^{\varepsilon-1} d_* p^j\right)$
, where $d_0 \in \mathbb{Z}_p$, and $d_* \in \mathbb{I}$, and ε' is z_i's least significant base-p digit index whose value is non-zero after digit index 0. Therefore, each $z_i \in \mathbb{Z}_{p^\varepsilon}$ is mapped to a unique set of (d_0, d_*, ε').

Next, we define a *lifting* polynomial $F_{\varepsilon'}$ in terms of z_i and its associated (d_0, d_*, ε') as follows:
$F_{\varepsilon'}(z_i) \equiv d_0 \bmod p^{\varepsilon'+1}$

Verbally speaking, $F_{\varepsilon'}(z_i)$ processes z_i in such a way that it keeps $z_{i,0}$ (i.e., z_i's value at the base-p digit index 0) the same as before, then finds the next least significant base-p digit whose value is non-zero (whose digit index is denoted as ε') and zeros it, during which the subsequent higher significant base-p digits may get updated to any arbitrary values (i.e., the function doesn't care about those values whose base-p digit index is higher than ε' because they fall outside the modulo $p^{\varepsilon'+1}$ range).

We will show an example of how z_i is updated if it is evaluated by the $F_{\varepsilon'}$ function recursively total $\varepsilon - 1$ times in a row as follows:
$\underbrace{F_{\varepsilon-1} \cdots F_3 \circ F_2 \circ F_1}_{\varepsilon-1 \text{ times}}(z_i)$

1st Recursion: $F_1(z_i) = c_{i,\varepsilon-1}p^{\varepsilon-1} + c_{i,\varepsilon-2}p^{\varepsilon-2} + \cdots + c_{i,2}p^2 + 0p + z_{i,0} \bmod p^\varepsilon$ # $F_1(z_i) \equiv z_{i,0} \bmod p^2$

2nd Recursion: $F_2 \circ F_1(z_i) = c'_{i,\varepsilon-1}p^{\varepsilon-1} + c'_{i,\varepsilon-2}p^{\varepsilon-2} + \cdots + 0p^2 + 0p + z_{i,0} \bmod p^{\varepsilon}$ # $F_1 \circ F_2(z_i) \equiv z_{i,0} \bmod p^3$

3rd Recursion: $F_3 \circ F_2 \circ F_1(z_i) = c''_{i,\varepsilon-1}p^{\varepsilon-1} + c''_{i,\varepsilon-2}p^{\varepsilon-2} + \cdots + 0p^3 + 0p^2 + 0p + z_{i,0} \bmod p^{\varepsilon}$ # $F_3 \circ F_2 \circ F_1(z_i) \equiv z_{i,0} \bmod p^4$

$\vdots$

$(\varepsilon-1)$-th Recursion: $\underbrace{F_{\varepsilon-1} \circ \cdots \circ F_3 \circ F_2 \circ F_1}_{\varepsilon-1 \text{ times}}(z_i) = 0p^{\varepsilon-1} + 0p^{\varepsilon-2} + \cdots + 0p^2 + 0p + z_{i,0} \bmod p^{\varepsilon}$ # $F_{\varepsilon-1} \cdots F_3 \circ F_2 \circ F_1(z_i) \equiv z_{i,0} \bmod p^{\varepsilon}$

In the above recursive computation, notice that the order of using function $F_{\varepsilon'}$ is specifically $F_1 \to F_2 \to F_3 \to \cdots \to F_{\varepsilon-1}$. We choose this specific order because we assume that for the initial input z_i, we don't know its associated ε' value (i.e., the least significant base-p digit index whose value is non-zero after digit index 0). If we choose the order $F_1 \to F_2 \to F_3 \to \cdots \to F_{\varepsilon-1}$, then regardless of the value of z_i, we get the universal guarantee that the final output will be $z_{i,0} \bmod p^{\varepsilon}$ (i.e., the value of the base-p digit index 0).

Now, we define the digit extraction function $G_{\varepsilon,v}(z_i)$ as follows:
$G_{\varepsilon,v}(z_i) = z_i - (\underbrace{F_{\varepsilon-1} \circ F_{\varepsilon-2} \circ F_{\varepsilon-3} \cdots \circ F_{\varepsilon-v}}_{v \text{ times}}(z_i) \bmod p^{\varepsilon}$

Notice that $G_{\varepsilon,v}(z_i)$ is equivalent to zeroing out the least significant base-p digit of z_i. Let's see what happens if we recursively evaluate $G_{\varepsilon,v}$ at z_i for $v = \varepsilon-1, \varepsilon-2, \cdots, 1$ (total $\varepsilon-1$ times):
1st Recursion: $G_{\varepsilon,\varepsilon-1}(z_i) = z_{i,\varepsilon-1}p^{\varepsilon-1} + z_{i,\varepsilon-2}p^{\varepsilon-2} + \cdots + z_{i,2}p^2 + z_{i,1}p + 0 \bmod p^{\varepsilon}$
2nd Recursion: $G_{\varepsilon,\varepsilon-2} \circ G_{\varepsilon,\varepsilon-1}(z_i) = z_{i,\varepsilon-1}p^{\varepsilon-1} + z_{i,\varepsilon-2}p^{\varepsilon-2} + \cdots + z_{i,2}p^2 + 0p + 0 \bmod p^{\varepsilon}$
3rd Recursion: $G_{\varepsilon,\varepsilon-3} \circ G_{\varepsilon,\varepsilon-2} \circ G_{\varepsilon,\varepsilon-1}(z_i) = z_{i,\varepsilon-1}p^{\varepsilon-1} + z_{i,\varepsilon-2}p^{\varepsilon-2} + \cdots + 0p^2 + 0p + 0 \bmod p^{\varepsilon}$

$\vdots$

$\varepsilon-1$-th Recursion: $G_{\varepsilon,1} \circ \cdots \circ G_{\varepsilon,\varepsilon-2} \circ G_{\varepsilon,\varepsilon-1}(z_i) = z_{i,\varepsilon-1}p^{\varepsilon-1} + 0p^{\varepsilon-2} + \cdots + 0p + 0 \bmod p^{\varepsilon}$

As shown above, recursively evaluating $G_{\varepsilon,v}$ at z_i for $v = \varepsilon-1, \varepsilon-2, \cdots, 1$ (total $\varepsilon-1$ times) is equivalent to zeroing out the least significant (base-p) $\varepsilon-1$ digits modulo p^{ε}.

By recursively applying the digit extraction function $G_{\varepsilon,v}$ as above, we can zero out the noise stored at the least significant (base-p) $\varepsilon-1$ digit indices.
Now, our final remaining task is to design the actual *lifting* polynomial $F_{\varepsilon'}(z_i)$ that implements $G_{\varepsilon,v}$.

Designing $F_{\varepsilon'}(z_i)$: We will derive $F_{\varepsilon'}(z_i)$ based on the following steps.

1. Let's define $z_i = z_{i,0} + \sum_{j=\varepsilon'}^{\varepsilon-1} z_{i,j}p^j = z_{i,0} + kp^{\varepsilon'}$
   , where $z_{i,0} \in [0, p-1]$, ε' is the least significant base-p digit's index of z_i which has a non-zero value, and $k = p^{-\varepsilon'} \cdot \sum_{j=\varepsilon'}^{\varepsilon-1} z_{i,j}p^j \in [0, p^{\varepsilon-\varepsilon'} - 1]$.

Basically, z_i can be any number in $\mathbb{Z}_{p^\varepsilon}$ whose base-p representation has 0s between the base-p digit index greater than 0 and smaller than ε'.

2. ***Claim:*** $z_i^p \equiv z_{i,0} \bmod p$

   Proof. Fermat's Little Theorem states $a^{p-1} \equiv 1 \bmod p$ for all $a \in \mathbb{Z}_p$ and prime p. Therefore, $z_i \equiv z_{i,0}^p \bmod p$. □

3. ***Claim:*** $z_i^p \equiv z_{i,0}^p \bmod p^{\varepsilon'+1}$

   Proof.

   $$(z_{i,0} + kp^{\varepsilon'})^p \bmod p^{\varepsilon'+1} = \sum_{j=0}^{p} \binom{p}{j} \cdot z_{i,0}^j \cdot (kp^{\varepsilon'})^{p-j} \bmod p^{\varepsilon'+1} \text{\# binomial expansion formula}$$

   $$= z_{i,0}^p$$ □

4. ***Claim:*** Given p and ε' are fixed, there exists $\varepsilon'+1$ polynomials $f_0, f_1, f_2, \cdots, f_{\varepsilon'}$ (where each polynomial is at most $p-1$ degrees) such that any z_i (i.e., any number whose base-p representation has 0s between the base-p digit index greater than 0 and smaller than ε') can be expressed as the following formula:
   $z_i^p \equiv \sum_{j=0}^{\varepsilon'} f_j(z_{i,0}) \cdot p^j \bmod p^{\varepsilon'+1}$

   Proof.
   $z_i^p \bmod p^{\varepsilon'+1}$ can be expressed as a base-p number as follows:
   $z_i^p \bmod p^{\varepsilon'+1} = c_0 + c_1 p + c_2 p^2 + \cdots + c_{\varepsilon'} p^{\varepsilon'}$

   Based on step 3's claim ($z_i^p \equiv z_{i,0}^p \bmod p^{\varepsilon'+1}$), we know that the value of $z_i^p \bmod p^{\varepsilon'+1}$ depends only on $z_{i,0}$ (given p and ε' are fixed). Therefore, we can imagine that there exists some function $f(z_{i,0})$ whose input is $z_{i,0} \in [0, p-1]$ and the output is $z_i^p \in [0, p^{\varepsilon'+1} - 1]$. Alternatively, we can imagine that there exist $\varepsilon'+1$ different functions $f_0, f_1, \cdots, f_{\varepsilon'}$ such that each f_j is a polynomial whose input is $z_{i,0} \in [0, p-1]$ and the output is $c_i \in [0, p-1]$, and $z_i^p \equiv \sum_{j=0}^{\varepsilon'} f_j(z_{i,0}) \cdot p^j \bmod p^{\varepsilon'+1}$.
   The input and output domain of each polynomial f_j is $[0, p-1]$. Therefore, we can design each f_j as a $(p-1)$-degree polynomial and derive each f_j based on polynomial interpolation (**??**) by using $(p-1)$ coordinate values.
   Note that whenever we increase ε' to $\varepsilon'+1$, we add a new polynomial $f_{\varepsilon'+1}$. However, the previous polynomials $f_0, f_1, \cdots, f_{\varepsilon'}$ stay the same as before, because increasing ε' by 1 only adds a new base-p constant $c_{\varepsilon'+1}$ for the highest base-p digit, while keeping the lower-digit constants $c_0, c_1, \cdots, c_{\varepsilon'}$ the same as before. Therefore, the polynomials $f_0, f_1, \cdots, f_{\varepsilon'}$, each of which computes $c_0, c_1, \cdots, c_{\varepsilon'}$, also stay the same as before.

   □

5. ***Claim:*** The formula in step 4's claim can be further concretized as follows:
   $z_i^p \equiv z_{i,0} + \sum_{j=1}^{\varepsilon'} f_j(z_{i,0}) \cdot p^j \bmod p^{\varepsilon'+1}$

Proof.
According to step 2's claim ($z_i^p \equiv z_{i,0} \bmod p$), we know that the following base-p representation of z_i^p:
$z_i^p \bmod p^{\varepsilon'+1} = c_0 + c_1 p + c_2 p^2 + \cdots + c_{\varepsilon-1} p^{\varepsilon'}$

will be the following:
$z_i^p \bmod p^{\varepsilon'+1} = z_{i,0} + c_1 p + c_2 p^2 + \cdots + c_{\varepsilon-1} p^{\varepsilon'}$

, since the least significant base-p digit of z_i^p in the base-p representation is always $z_{i,0}$. Thus, the formula in step 4's claim:
$z_i^p = \sum_{j=0}^{\varepsilon'} f_j(z_{i,0}) \cdot p^j \bmod p^{\varepsilon'+1}$

can be further concretized as follows:

$z_i^p = z_{i,0} + \sum_{j=1}^{\varepsilon'} f_j(z_{i,0}) \cdot p^j \bmod p^{\varepsilon'+1}$

□

6. ***Claim:*** $z_i^p - \sum_{j=1}^{\varepsilon'} f_j(z_i) \cdot p^j \equiv z_{i,0} \bmod p^{\varepsilon'+1}$

Proof.

Remember that in step 1, we defined z_i as: $z_i = z_{i,0} + \sum_{j=\varepsilon'}^{\varepsilon-1} z_{i,j} p^j$. Therefore, $z_i \bmod p^{\varepsilon'} = z_{i,0}$. This implies that for each polynomial f_j, the following is true:
$f_j(z_{i,0}) \equiv f_j(z_i) \bmod p^{\varepsilon'}$

We can further derive the following:
$f_j(z_{i,0}) \cdot p^j \equiv f_j(z_i) \cdot p^j \bmod p^{\varepsilon'+1}$ (where $i \geq 1$)

The above is true because:
$f_j(z_{i,0}) \equiv f_j(z_i) \bmod p^{\varepsilon'}$
$f_j(z_{i,0}) = f_j(z_i) + q \cdot p^{\varepsilon'}$ (for some integer q)
$f_j(z_{i,0}) \cdot p^j = f_j(z_i) \cdot p^j + q \cdot p^{\varepsilon'} \cdot p^j$ # multiplying p^j to both sides
$f_j(z_{i,0}) \cdot p^j = f_j(z_i) \cdot p^j + q \cdot p^{j-1} \cdot p^{\varepsilon'+1}$ # $f_j(z_{i,0}) \cdot p^j$ and $f_j(z_i) \cdot p^j$ differ by some multiple of $p^{\varepsilon'+1}$

Therefore, $f_j(z_{i,0}) \cdot p^j \equiv f_j(z_i) \cdot p^j \bmod p^{\varepsilon'+1}$.

Now, given step 5's claim ($z_i^p \equiv z_{i,0} + \sum_{j=1}^{\varepsilon'} f_j(z_i) \cdot p^j \bmod p^{\varepsilon'+1}$), we can derive the following:

$z_i^p - \sum_{j=1}^{\varepsilon'} f_j(z_i) \cdot p^j \bmod p^{\varepsilon'+1}$
$\equiv (z_{i,0} + \sum_{j=1}^{\varepsilon'} f_j(z_{i,0}) \cdot p^j) - \sum_{j=1}^{\varepsilon'} f_j(z_i) \cdot p^j \bmod p^{\varepsilon'+1}$ # applying step 5's claim

$$\equiv (z_{i,0} + \sum_{j=1}^{\varepsilon'} f_j(z_i) \cdot p^j) - \sum_{j=1}^{\varepsilon'} f_j(z_i) \cdot p^j \bmod p^{\varepsilon'+1} \text{ \# since } f_j(z_{i,0}) \cdot p^j = f_j(z_i) \cdot p^j \bmod p^{\varepsilon'+1}$$
$$\equiv z_{i,0} \bmod p^{\varepsilon'+1}$$

□

7. Finally, we define the lifting polynomial $F_{\varepsilon'}(z_i)$ as follows:

$$F_{\varepsilon'}(z_i) = z_i^p - \sum_{j=1}^{\varepsilon'} f_j(z_i) \cdot p^j$$
$$\equiv z_{i,0} \bmod p^{\varepsilon'+1}$$

The above relation implies that $F_{\varepsilon'}(x) \bmod p^{\varepsilon'+1}$ is equivalent to the least significant base-p digit of x (according to step 6's claim). Therefore, if we plug in z_i into $F_{\varepsilon'}(x)$ and regard $\varepsilon' = 1$, then the output is some number whose least significant base-p digit is $z_{i,0} \bmod p^2$ and the 2nd least significant base-p digit is $0 \bmod p^2$. As we recursively apply the output back to $F_{\varepsilon'}(x)$ and increment ε' by 1, we iteratively zero out the 2nd least significant base-p digit, the 3rd least significant base-p digit, and so on. We repeat this process for $\varepsilon - 1$ times to zero out the upper $\varepsilon - 1$ base-p digits, keeping only the least significant digit as it is (i.e., $z_{i,0}$). Therefore, $F_{\varepsilon'}(x)$ is a valid lifting polynomial that can be iteratively used to extract the least significant digit of $z_i \bmod p^{\varepsilon}$.

We use $F_{\varepsilon'}(x)$ as the internal helper function within the digit extraction function $G_{\varepsilon,v}(z_i)$ that calls $F_{\varepsilon'}(x)$ a total $v - 1$ times.

D-2.11.6 Scaling Factor Re-interpretation

Remember that homomorphically decrypting (A', B') outputs an encryption of $p^{\varepsilon-1}M + E' + Kp^{\varepsilon}$, whose entire value is bigger than p^{ε}. The final result of homomorphic digit extraction is equivalent to zeroing out the least significant (base-p) $\varepsilon - 1$ digits of the input value $p^{\varepsilon-1}M + E' + Kp^{\varepsilon}$. Therefore, at the end of digit extraction, we get a ciphertext that encrypts $p^{\varepsilon-1}M + K'p^{\varepsilon}$, where $K'p^{\varepsilon}$ is a polynomial with coefficients which are some multiples of p to account for the wrapping values of p^{ε}.

Therefore, the output of the digit extraction and $\mathsf{SlotToCoeff}$ steps is the ciphertext $\mathsf{RLWE}_{S,\sigma}(\Delta' \cdot (p^{\varepsilon-1}M + K'p^{\varepsilon})) \bmod q$, where the plaintext modulus is p^{ε} and the plaintext scaling factor $\Delta' = \frac{q}{p^{\varepsilon}}$ (as we derived in **??**). However, our desired outcome of the BFV bootstrapping is a noise-removed ciphertext whose plaintext modulus is p and the plaintext scaling factor is $\Delta = \frac{q}{p}$. Without doing any actual additional computation, we can view the above ciphertext $\mathsf{RLWE}_{S,\sigma}(\Delta' \cdot (p^{\varepsilon-1}M + K'p^{\varepsilon})) \bmod q$ as an encryption of plaintext modulo p whose scaling factor is $\Delta = \frac{q}{p}$. This is because:

$$\mathsf{RLWE}_{S,\sigma}(\Delta' \cdot (p^{\varepsilon-1}M + K'p^{\varepsilon})) = \mathsf{RLWE}_{S,\sigma}\left(\frac{q}{p^{\varepsilon}} \cdot (p^{\varepsilon-1}M + K'p^{\varepsilon})\right)$$
$$= \mathsf{RLWE}_{S,\sigma}\left(\frac{q}{p}M + K'q\right)$$
$$= \mathsf{RLWE}_{S,\sigma}\left(\frac{q}{p}M\right) \bmod q$$

Therefore, we can view the above ciphertext as $\mathsf{RLWE}_{S,\sigma}(\Delta M) \bmod q$ whose plaintext modulus is

p and scaling factor $\Delta = \frac{q}{p}$, because the underlying ciphertext's structure of these two viewpoints is identical. This leads to the corollary that evaluating the digit extraction function (**??**) at z_i recursively total $\varepsilon - 1$ times is logically equivalent to the noise elimination and plaintext modulus switch ($p^\varepsilon \to p$) as follows:

$$G_{\varepsilon,1} \circ \cdots \circ G_{\varepsilon,\varepsilon-2} \circ G_{\varepsilon,\varepsilon-1}(z_i) = \left\lfloor \frac{z_i}{p^{\varepsilon-1}} \right\rfloor \bmod p$$

We can optionally design a more precise rounding-based function by modifying z_i to $z_i + \frac{p^{\varepsilon-1}}{2}$ as follows:

$$G_{\varepsilon,1} \circ \cdots \circ G_{\varepsilon,\varepsilon-2} \circ G_{\varepsilon,\varepsilon-1}\left(z_i + \frac{p^{\varepsilon-1}}{2}\right) = \left\lceil \frac{z_i}{p^{\varepsilon-1}} \right\rfloor \bmod p$$

Generalization of $t = p^r$: We have explained the BFV bootstrapping with the assumption that the plaintext modulus $t = p$ is a prime number. However, we can generalize t as $t = p^r$ where r can be any positive integer. The benefit of choosing $t = p^r$ with a small p instead of $t = p$ with a big p is the efficient noise management of the digit extraction process. As digit extraction requires many ciphertext-to-plaintext multiplications with $p, p^2, \cdots, p^{\varepsilon-1}$, using a small p generates a smaller noise during the homomorphic operations.

D-2.11.7 Summary

We summarize the BFV bootstrapping procedure (with the generalization of $t = p^r$) as follows.

⟨Summary ??⟩ BFV Bootstrapping

Suppose we have an RLWE ciphertext $(A, B) = \mathsf{RLWE}_{S,\sigma}(\Delta M + E) \bmod q$, where $\Delta = \left\lfloor \frac{q}{t} \right\rfloor$ and $t = p^r$ (i.e., the plaintext modulus is a power of some prime), $r \in \mathbb{I}$, and $r \geq 1$.

1. **Modulus Switch (from $q \to p^\varepsilon$):** Scale down the ciphertext from (A, B) to $\left(\left\lceil \frac{p^\varepsilon}{q} \cdot A \right\rfloor, \left\lceil \frac{p^\varepsilon}{q} \cdot B \right\rfloor\right) = (A', B')$ # where $p^\varepsilon \ll q$

   $A'S + B' = p^{\varepsilon-r}M + E' \bmod p^\varepsilon$ # where $E' \approx \frac{p^\varepsilon}{q} \cdot E + \left(\left\lfloor \frac{q}{p} \right\rfloor \cdot \frac{p^\varepsilon}{q} - p^{\varepsilon-r}\right) \cdot M$, which is a modulus switch noise plus a rounding noise caused by treating $\Delta = \left\lfloor \frac{q}{p^r} \right\rfloor \approx \frac{q}{p^r}$.

2. **Homomorphic Decryption:** With the bootstrapping key $\mathsf{RLWE}_{S,\sigma}(\Delta' S) \bmod q$, homomorphically decrypt $(A', B') \bmod p^\varepsilon$ as follows:

   $A' \cdot \mathsf{RLWE}_{S,\sigma}(\Delta' S) + B' = \mathsf{RLWE}_{S,\sigma}(\Delta' \cdot (p^{\varepsilon-r}M + E' + Kp^\varepsilon)) \bmod q$ # where $\Delta' = \left\lfloor \frac{q}{p^\varepsilon} \right\rfloor$

   Now, we denote the modulus-switched noisy plaintext polynomial as $Z = p^{\varepsilon-r}M + E' + Kp^\varepsilon$.

3. **CoeffToSlot:** Move the (encrypted) polynomial Z's coefficients $z_0, z_i, \cdots, z_{n-1}$ to the input vector slots. This is done by computing:

$\mathsf{RLWE}_{S,\sigma}(\Delta' Z) \cdot n^{-1} \cdot \tilde{W} \cdot I_R^n$
$= \mathsf{RLWE}_{S,\sigma}(\Delta' Z^{\langle 1 \rangle})$
, where $n^{-1} \cdot \tilde{W} \cdot I_R^n$ is the batch encoding matrix (Summary **??** in **??**).

4. **Digit Extraction:** We design a polynomial $G_{\varepsilon,v}(z_i)$ (a digit extraction polynomial) as follows:
$z_i = d_0 + \left(\sum_{j=\varepsilon'}^{\varepsilon - r} d_* p^j \right)$ # where $d_0 \in \mathbb{Z}_{p^r}$, and ε' is z_i's least significant base-p digit index whose value is non-zero after digit index after digit index $r-1$
$F_{\varepsilon'}(z_i) \equiv d_0 \bmod p^{\varepsilon'+1}$ # a $(p-1)$-degree polynomial recursively used to finally extract the value $d_0 \bmod p^{\varepsilon}$
$G_{\varepsilon,v}(z_i) \equiv z_i - \underbrace{F_{\varepsilon-1} \circ F_{\varepsilon-2} \circ F_{\varepsilon-3} \cdots F_1}_{\varepsilon-1 \text{ times}}(z_i) \bmod p^{\varepsilon}$

   We homomorphically evaluate the digit extraction polynomial $G_{\varepsilon,v}$ for $v = \{\varepsilon - r - 1, \varepsilon - r - 2, \cdots, 1\}$ recursively total $\varepsilon - r - 1$ times at each coefficient z_i of Z stored at input vector slots, which zeros out the least significant (base-p) $\varepsilon - r - 1$ digits of z_i as follows:
$G_{\varepsilon,1} \circ G_{\varepsilon,2} \circ \cdots \circ G_{\varepsilon,\varepsilon-r-2} \circ G_{\varepsilon,\varepsilon-r-r}(z_i) \bmod p^{\varepsilon}$
$= p^{\varepsilon - r} m_i + k_i' p^{\varepsilon}$

   , provided E''s each coefficient $\varepsilon_i' < \dfrac{p^{\varepsilon - r}}{2}$. At this point, each input vector slot contains the noise-removed coefficient $p^{\varepsilon - r} m_i + k_i' p^{\varepsilon}$.

5. **SlotToCoeff:** Homomorphically move each input vector slot's value $p^{\varepsilon - r} m_i + k_i p^{\varepsilon}$ back to the (encrypted) polynomial coefficient positions. This is done by multiplying $\tilde{W}^*$ to the output ciphertext of the digit extraction step, where $\tilde{W}^*$ is the decoding matrix (Summary **??** in **??**). The output of this computation is $\mathsf{RLWE}_{S,\sigma}(\Delta' \cdot (p^{\varepsilon - r} M + K' p^{\varepsilon})) \bmod q$, where $\Delta' = \dfrac{q}{p^{\varepsilon}}$ and the plaintext modulus is p^{ε}.

6. **Scaling Factor Re-interpretation:**
   The output of the previous step is $\mathsf{RLWE}_{S,\sigma}(\Delta' \cdot (p^{\varepsilon - r} M + K' p^{\varepsilon})) \bmod q$, where $\Delta' = \dfrac{q}{p^{\varepsilon}}$, which is a modulo-$q$ ciphertext encrypting a modulo-p^{ε} plaintext with the scaling factor $\dfrac{q}{p^{\varepsilon}}$. Without any additional computation, we can theoretically view this ciphertext as a modulo-q ciphertext encrypting a modulo-p plaintext with the scaling factor $\dfrac{q}{p}$. This is because:
$\mathsf{RLWE}_{S,\sigma}(\Delta' \cdot (p^{\varepsilon - r} M + K' p^{\varepsilon})) = \mathsf{RLWE}_{S,\sigma}\left(\dfrac{q}{p^{\varepsilon}} \cdot (p^{\varepsilon - r} M + K' p^{\varepsilon}) \right)$
$= \mathsf{RLWE}_{S,\sigma}\left(\dfrac{q}{p} M + K' q \right)$
$= \mathsf{RLWE}_{S,\sigma}\left(\dfrac{q}{p} M \right) \bmod q$

   For these two viewpoints, their underlying ciphertext structure is identical. Therefore,

the digit extraction function (**??**) at z_i recursively total $\varepsilon - r$ times is logically equivalent to the noise elimination and plaintext modulus switch ($p^{\varepsilon} \to p$) as follows:

$$G_{\varepsilon,1} \circ \cdots \circ G_{\varepsilon,\varepsilon-r-2} \circ G_{\varepsilon,\varepsilon-r-1}(z_i) = \left\lfloor \frac{z_i}{p^{\varepsilon-r}} \right\rfloor \bmod p$$

A more precise rounding-based logic can be designed by modifying z_i to $z_i + \frac{p^{\varepsilon-r}}{2}$ as follows:

$$G_{\varepsilon,1} \circ \cdots \circ G_{\varepsilon,\varepsilon-r-2} \circ G_{\varepsilon,\varepsilon-r-1} \left(z_i + \frac{p^{\varepsilon-r-1}}{2} \right) = \left\lceil \frac{z_i}{p^{\varepsilon-r}} \right\rfloor \bmod p$$

Necessity of Homomorphic Decryption: Suppose that we performed the BFV bootstrapping without homomorphic decryption. Then, the input to the digit extraction step would be $p^{\varepsilon-r}M + Kp^{\varepsilon}$, not $p^{\varepsilon-r}M + E' + Kp^{\varepsilon}$. This is because the ciphertext (A', B') encrypts $\mathsf{RLWE}_{S,\sigma}(\Delta'' M)$, where $\Delta'' = p^{\varepsilon-r}$. Therefore, applying the $\mathsf{CoeffToSlot}$ transformation to $\mathsf{RLWE}_{S,\sigma}(\Delta'' M)$ will store the coefficients of $M \bmod p^{\varepsilon}$, not the coefficients of $\Delta'' M + E' = p^{\varepsilon-r}M + E' \bmod p^{\varepsilon}$. In order to preserve the noise E' as well, we homomorphically decrypt $\mathsf{RLWE}_{S,\sigma}(\Delta'' M)$ by using an encrypted secret key $\mathsf{RLWE}_{S,\sigma}(\Delta' S)$ to make it $\mathsf{RLWE}_{S,\sigma}(\Delta' \cdot (p^{\varepsilon-r}M + E' + Kp^{\varepsilon}))$.

D-3 CKKS Scheme

The CKKS scheme is designed for homomorphic addition and multiplication of complex numbers that contain imaginary numbers. Therefore, unlike BFV, BGV, or TFHE, which can only compute over integers, CKKS can compute real-world floating point arithmetic, such as machine learning.

The CKKS scheme's goal is to homomorphically compute addition and multiplication of complex numbers. However, while our targeted inputs are complex numbers, CKKS's plaintext space is defined as a $(n-1)$-degree polynomial ring with real-number coefficients having a limited precision, that is, $\mathcal{R}_{\langle n \rangle} = \mathbb{R}[x]/(x^n+1)$. Therefore, CKKS designs its unique encoding scheme which converts the input complex numbers into integers which can be used as coefficients of a polynomial in $\mathcal{R}_{\langle n \rangle}$.

In overall, CKKS's encryption procedure is as follows:
1. $\underline{\mathsf{Encoding}_1}$: Encode the targeted input complex number into a real number
2. $\underline{\mathsf{Encoding}_2}$: Encode the real number into an integer
3. $\underline{\mathsf{Encryption}}$: Encrypt the integer by RLWE

The encrypted RLWE ciphertext supports homomorphic addition and multiplication.

At the end of all homomorphic operations, CKKS's decryption procedure is as follows:
1. $\underline{\mathsf{Decryption}}$: Decrypt the RLWE ciphertext into a plaintext integer
2. $\underline{\mathsf{Decoding}_1}$: Decode the integer to a real number
3. $\underline{\mathsf{Decoding}_2}$: Decode the real number to a complex number

Remember that BFV is an exact encryption scheme based on rings. On the other hand, CKKS introduces a drifting error while its encoding process of rounding square-root values (included in the Euler's formula) to the nearest integer. Therefore, its decryption is not exactly the same as before encryption. Such a small error occurring during encryption and decryption makes CKKS an *approximate* encryption scheme.

CKKS internally uses the same schemes as BFV for encryption, decryption, ciphertext-to-plaintext addition, ciphertext-to-ciphertext addition, and ciphertext-to-plaintext multiplication. Meanwhile, CKKS uses slightly different schemes than BFV for encoding the input vector (i.e., input vector slots) rotation (if BFV uses the batch encoding scheme), ciphertext-to-ciphertext multiplication, and bootstrapping. This difference comes from the fact that CKKS handles homomorphic operations over complex numbers as inputs, whereas BFV handles homomorphic operations over rings.

Required Background

- ??: ??
- ??: ??
- ??: ??
- ??: ??
- ??: ??
- ??: ??
- ??: ??
- ??: ??
- ??: ??

- ??: ??
- ??: ??
- ??: ??
- ??: ??
- ??: ??
- ??: ??
- ??: ??
- ??: ??
- ??: ??
- ??: ??
- ??: ??
- ??: ??
- ??: ??
- ??: ??
- ??: ??
- ??: ??

D-3.1 Encoding and Decoding

CKKS's encoding and decoding is fundamentally very similar to BFV's batch encoding scheme. BFV designs its batch encoding scheme (Summary **??** in **??**) based on the updated $\tilde{W}$ and $\tilde{W}^*$ matrices (Summary **??** in **??**). That is, BFV decodes a polynomial into an input slot vector by evaluating the polynomial at each root of X^n+1, which is the primitive ($\mu = 2n$)-th root of unity (i.e., $\vec{v} = \tilde{W}^* \cdot \vec{m}$), and encodes an input slot vector into a polynomial by inversing this operation (i.e., $\vec{m} = n^{-1} \cdot \tilde{W} \cdot I_n^R \cdot \vec{v}$). This encoding and decoding scheme is designed based on Summary **??** (**??**) which designs the isomorphic mapping between n-dimensional vectors in a ring (finite field) and $(n-1)$-degree (or lesser degree) polynomials as follows:

$\sigma : f(x) \in \mathbb{Z}_t[X]/F(X) \longrightarrow (f(\omega^1), f(\omega^3)), \cdots, f(\omega^{2n-1})) \in \mathbb{Z}_t^n$

, where $\omega = g^{\frac{t-1}{2n}}$ is a root of (i.e., primitive ($\mu = 2n$)-th root of unity) of the ($\mu = 2n$)-th cyclotomic polynomial X^n+1 defined over a prime modulo t ring.

CKKS's batch encoding scheme uses exactly the same formula for encoding and decoding (i.e., $\vec{v} = W^T \cdot \vec{m}$ and $\vec{m} = \frac{W \cdot I_n^R \cdot \vec{v}}{n}$), but the n-dimensional input slot vector comprises not in a ring (i.e., $\mathbb{Z}_p^n$), but complex numbers (i.e., $\hat{\mathbb{C}}^n$). In Summary **??** (**??**), we also designed the mapping σ_c between polynomials and vectors over complex numbers as follows:

$\sigma_c : f(X) \in \mathbb{R}[X]/(X^n+1) \longrightarrow (f(\omega), f(\omega^3), f(\omega^5), \cdots, f(\omega^{2n-1})) \in \hat{\mathbb{C}}^n \ (\longrightarrow \mathbb{C}^{\frac{n}{2}})$

, where $\omega = e^{i\pi/n}$ is a root (i.e., the primitive ($\mu = 2n$)-th root) of the ($\mu = 2n$)-th cyclotomic polynomial X^n+1 defined over complex numbers, and $\hat{\mathbb{C}}^n$ is n-dimensional complex special vector space whose second-half elements are reverse-ordered conjugates of the first-half elements. And $\hat{\mathbb{C}}^n$ is isomorphic to $\mathbb{C}^{\frac{n}{2}}$, because the second-half elements of $\hat{\mathbb{C}}^n$ are automatically determined by its first-half elements. Therefore, the σ_c mapping is essentially an isomorphism between $\frac{n}{2}$-dimensional complex vectors $\vec{v} \in \mathbb{C}^{\frac{n}{2}}$ and $(n-1)$-degree (or lesser degree) real-number polynomials $\mathbb{R}[X]/(X^n+1)$. Therefore, CKKS' batch encoding scheme encodes an $\frac{n}{2}$-dimensional complex input slot vector into an $(n-1)$-degree (or lesser degree) real-number polynomial, and the decoding process is a reverse of this.

In addition, remember that in BFV, we updated W and W^T to $\tilde{W}$ and $\tilde{W}^*$ (Summary **??** in **??**) to support homomorphic rotation of input vector slots. Likewise, the CKKS batch encoding scheme uses $\tilde{W}$ and $\tilde{W}^*$ instead of W and W^T in order to support homomorphic rotation. Therefore, the CKKS batch encoding scheme's isomorphic mapping is updated as follows:

$$\sigma_c : f(X) \in \mathbb{R}[X]/(X^n+1) \longrightarrow \big(f(\omega^{J(0)}), f(\omega^{J(1)}), f(\omega^{J(2)}), \cdots, f(\omega^{J(\frac{n}{2}-1)}),$$
$$\cdots, f(\omega^{J_*(0)}), f(\omega^{J_*(1)}), f(\omega^{J_*(2)}), \cdots, f(\omega^{J_*(\frac{n}{2}-1)})\big) \in \hat{\mathbb{C}}^n \ (\longrightarrow \mathbb{C}^{\frac{n}{2}})$$

, where $J(h) = 5^h \bmod 2n$, a rotation helper formula.

The encoding schemes of BFV and CKKS have the following differences:

- **Type of Input Slot Values:** BFV's input slot values are n-dimensional integers modulo t, which are encoded into n-dimensional polynomial coefficients (i.e., modulo-t integers). On the other hand, CKKS's input slot values are $\frac{n}{2}$-dimensional complex numbers, which are encoded into n-dimensional polynomial coefficients (i.e., real numbers).
- **Type of Polynomial Coefficients:** BFV's encoded polynomial coefficients are integer moduli, whereas CKKS's encoded polynomial coefficients are real numbers.

- **Scaling Factor:** Both BFV and CKKS scales their encoded polynomial coefficients $\vec{m}$ by Δ to $\lceil \Delta \cdot \vec{m} \rfloor$. BFV's suggested scaling factor is $\Delta = \lfloor \frac{q_0}{t} \rfloor$, but CKKS's scaling factor Δ has no suggested formula because its polynomial coefficients are real numbers not bound by modulus, and thus it can be any value provided that the scaled coefficients do not overflow or underflow the range $[1, q_0 - 1]$ (or $\left[-\frac{q_0}{2}, \frac{q_0}{2}\right)$).
- **Encoding Precision:** In the case of BFV, during its decoding process, BFV's down-scaled polynomial coefficients $\frac{\Delta \cdot \vec{m}}{\Delta}$ preserve the precision of input values . On the other hand, CKKS's down-scaled polynomial coefficients may lose their precision if their original input values have too many decimal digits so that the scaling factor cannot left-shift all of them to make them part of the integer domain, which means that some lower decimal digits of the input value may be rounded off, which loses precision of the original input. For example, suppose the polynomial coefficient $m_i = \frac{1}{3} = 0.33333\cdots$, and the scaling factor $\Delta = 100$. Then, the scaled coefficient $\lceil \Delta m_i \rfloor = 33$, and down-scaling it gives $\frac{33}{100} = 0.33$. Since $0.33 \neq 0.33333\cdots$, CKKS's encoding and decoding process does not always guarantee exact precision. Due to this encoding error, CKKS is called an *approximate* encryption scheme. The impact of this encoding error can grow over homomorphic operations which increases the magnitude of error and the decoded result would gradually become more deviated from the expected exact value. One way to reduce CKKS's encoding error is to increase Δ, and thereby left-shift more decimal digits to make them part of the scaled integer digits.

Structure of $\vec{v}' \in \hat{\mathbb{C}}^n$: Note that the original decoding scheme for $\vec{v}'$ described in Summary **??** (**??**) was:

$\vec{v}' = (\ M(\omega),\ M(\omega^3),\ M(\omega^5), \cdots, M(\omega^{2n-3}),\ M(\omega^{2n-1}))$

, which decodes to a Hermitian vector:

$\vec{v}' = (v_0, v_1, \cdots, v_{\frac{n}{2}-1}, \overline{v}_{\frac{n}{2}-1}, \cdots, \overline{v}_1, \overline{v}_0)$

, whose second-half elements are reverse-ordered conjugates of the first-half elements.

However, by replacing W and W^T with $\tilde{W}$ and $\tilde{W}^*$, we changed the above decoding scheme to the following that supports homomorphic rotation:

$\vec{v}' = (\ M(\omega^{J(0)}),\ M(\omega^{J(1)}),\ M(\omega^{J(2)}), \cdots, M(\omega^{J(\frac{n}{2}-1)}),\ M(\omega^{J_*(0)}),\ M(\omega^{J_*(1)}), \cdots, M(\omega^{J_*(\frac{n}{2}-1)})\ )$

$= (\ M(\omega^{J(0)}),\ M(\omega^{J(1)}),\ M(\omega^{J(2)}), \cdots, M(\omega^{J(\frac{n}{2}-1)}),\ M(\overline{\omega}^{J(0)}),\ M(\overline{\omega}^{J(1)}), \cdots, M(\overline{\omega}^{J(\frac{n}{2}-1)})\ )$

$\#$ because $\omega^{-1} = (e^{\frac{i\pi}{n}})^{-1} = e^{\frac{-i\pi}{n}} = \overline{\omega}$

, which decodes to a *forward-ordered* (not reverse-ordered) Hermitian vector as follows:

$\vec{v}' = (v_0, v_1, \cdots, v_{\frac{n}{2}-1}, \overline{v}_0, \overline{v}_1, \cdots, \overline{v}_{\frac{n}{2}-1},)$

, whose second-half elements are conjugates of the first-half elements with the same order. Upon homomorphic rotation (which will be explained in **??**),just like in BFV's homomorphic rotation, the first-half elements and the second-half elements of $\vec{v}'$ rotate within their own group in a wrapping manner.

We summarize CKKS's encoding and decoding procedure as follows, which is similar to BFV's encoding and decoding procedure (described in Summary **??** in **??**):

⟨Summary ??⟩ CKKS's Encoding and Decoding

Input: An $\frac{n}{2}$-dimensional complex vector $\vec{v} = (v_0, v_1, \cdots, v_{\frac{n}{2}-1}) \in \mathbb{C}^{\frac{n}{2}}$

Encoding:

1. Convert (i.e., isomorphically transform) $\vec{v}$ into an n-dimensional *forward-ordered* Hermitian vector $\vec{v}_r$ as follows:
   $\vec{v}_r = (v_0, v_1, \cdots, v_{\frac{n}{2}-1}, \overline{v}_0, \overline{v}_1, \cdots, \overline{v}_{\frac{n}{2}-1}) \in \hat{\mathbb{C}}^n$
2. Convert $\vec{v}_r$ into a real number vector $\vec{m}$ by applying the transformation $\vec{m} = \frac{\tilde{W} \cdot I_n^R \cdot \vec{v}_r}{n}$, where $\tilde{W}$ is a basis of the n-dimensional vector space crafted as follows:

$$\tilde{W} = \begin{bmatrix} 1 & 1 & \cdots & 1 & 1 & 1 & \cdots & 1 \\ (\omega^{J(\frac{n}{2}-1)}) & (\omega^{J(\frac{n}{2}-2)}) & \cdots & (\omega^{J(0)}) & (\omega^{J_*(\frac{n}{2}-1)}) & (\omega^{J_*(\frac{n}{2}-2)}) & \cdots & (\omega^{J_*(0)}) \\ (\omega^{J(\frac{n}{2}-1)})^2 & (\omega^{J(\frac{n}{2}-2)})^2 & \cdots & (\omega^{J(0)})^2 & (\omega^{J_*(\frac{n}{2}-1)})^2 & (\omega^{J_*(\frac{n}{2}-2)})^2 & \cdots & (\omega^{J_*(0)})^2 \\ \vdots & \vdots & \ddots & \vdots & \vdots & \ddots & \vdots & \vdots \\ (\omega^{J(\frac{n}{2}-1)})^{n-1} & (\omega^{J(\frac{n}{2}-2)})^{n-1} & \cdots & (\omega^{J(0)})^{n-1} & (\omega^{J_*(\frac{n}{2}-1)})^{n-1} & (\omega^{J_*(\frac{n}{2}-2)})^{n-1} & \vdots & (\omega^{J_*(0)})^{n-1} \end{bmatrix}$$

   # where $\omega = e^{i\pi/n} = \cos\left(\frac{\pi}{n}\right) + i \sin\left(\frac{\pi}{n}\right)$

$$= \begin{bmatrix} 1 & 1 & \cdots & 1 & 1 & 1 & \cdots & 1 \\ (\omega^{J(\frac{n}{2}-1)}) & (\omega^{J(\frac{n}{2}-2)}) & \cdots & (\omega^{J(0)}) & (\overline{\omega}^{J(\frac{n}{2}-1)}) & (\overline{\omega}^{J(\frac{n}{2}-2)}) & \cdots & (\overline{\omega}^{J(0)}) \\ (\omega^{J(\frac{n}{2}-1)})^2 & (\omega^{J(\frac{n}{2}-2)})^2 & \cdots & (\omega^{J(0)})^2 & (\overline{\omega}^{J(\frac{n}{2}-1)})^2 & (\overline{\omega}^{J(\frac{n}{2}-2)})^2 & \cdots & (\overline{\omega}^{J(0)})^2 \\ \vdots & \vdots & \ddots & \vdots & \vdots & \ddots & \vdots & \vdots \\ (\omega^{J(\frac{n}{2}-1)})^{n-1} & (\omega^{J(\frac{n}{2}-2)})^{n-1} & \cdots & (\omega^{J(0)})^{n-1} & (\overline{\omega}^{J(\frac{n}{2}-1)})^{n-1} & (\overline{\omega}^{J(\frac{n}{2}-2)})^{n-1} & \vdots & (\overline{\omega}^{J(0)})^{n-1} \end{bmatrix}$$

   # because $\omega^{-1} = e^{\frac{-i\pi}{n}} = \overline{e^{\frac{i\pi}{n}}} = \overline{\omega}$

3. Convert $\vec{m}$ into a scaled integer vector $\lceil \Delta\vec{m} \rfloor \approx \Delta\vec{m}$, where Δ is a scaling factor bigger than 1 such that Δm_i never overflows or underflows q_0 (i.e., $0 \leq \Delta m_i < q_0$ or $-\frac{q_0}{2} \leq \Delta m_i < \frac{q_0}{2}$) in all cases, even across all homomorphic operations. The finally encoded plaintext polynomial is $\Delta M = \sum_{i=0}^{n-1} \lceil \Delta m_i \rfloor X^i \in \mathbb{Z}_q[X]/(X^n + 1)$. The rounding process of $\lceil \Delta\vec{m} \rfloor$ during the encoding process causes an encoding error, which makes CKKS an approximate encryption scheme.

Decoding: From the plaintext polynomial $\Delta M = \sum_{i=0}^{n-1} \Delta m_i X^i$, recover $\vec{m} = \frac{\Delta\vec{m}}{\Delta}$. Then, compute $\vec{v}_r = \tilde{W}^* \cdot \vec{m}$, where:

$$\tilde{W}^* = \begin{bmatrix} 1 & (\omega^{J(0)}) & (\omega^{J(0)})^2 & \cdots & (\omega^{J(0)})^{n-1} \\ 1 & (\omega^{J(1)}) & (\omega^{J(1)})^2 & \cdots & (\omega^{J(1)})^{n-1} \\ 1 & (\omega^{J(2)}) & (\omega^{J(2)})^2 & \cdots & (\omega^{J(2)})^{n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & (\omega^{J(\frac{n}{2}-1)}) & (\omega^{J(\frac{n}{2}-1)})^2 & \cdots & (\omega^{J(\frac{n}{2}-1)})^{n-1} \\ 1 & (\omega^{J_*(0)}) & (\omega^{J_*(0)})^2 & \cdots & (\omega^{J_*(0)})^{n-1} \\ 1 & (\omega^{J_*(1)}) & (\omega^{J_*(1)})^2 & \cdots & (\omega^{J_*(1)})^{n-1} \\ 1 & (\omega^{J_*(2)}) & (\omega^{J_*(2)})^2 & \cdots & (\omega^{J_*(2)})^{n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & (\omega^{J_*(\frac{n}{2}-1)}) & (\omega^{J_*(\frac{n}{2}-1)})^2 & \cdots & (\omega^{J_*(\frac{n}{2}-1)})^{n-1} \end{bmatrix}$$

$$= \begin{bmatrix} 1 & (\omega^{J(0)}) & (\omega^{J(0)})^2 & \cdots & (\omega^{J(0)})^{n-1} \\ 1 & (\omega^{J(1)}) & (\omega^{J(1)})^2 & \cdots & (\omega^{J(1)})^{n-1} \\ 1 & (\omega^{J(2)}) & (\omega^{J(2)})^2 & \cdots & (\omega^{J(2)})^{n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & (\omega^{J(\frac{n}{2}-1)}) & (\omega^{J(\frac{n}{2}-1)})^2 & \cdots & (\omega^{J(\frac{n}{2}-1)})^{n-1} \\ 1 & (\overline{\omega}^{J(0)}) & (\overline{\omega}^{J(0)})^2 & \cdots & (\overline{\omega}^{J(0)})^{n-1} \\ 1 & (\overline{\omega}^{J(1)}) & (\overline{\omega}^{J(1)})^2 & \cdots & (\overline{\omega}^{J(1)})^{n-1} \\ 1 & (\overline{\omega}^{J(2)}) & (\overline{\omega}^{J(2)})^2 & \cdots & (\overline{\omega}^{J(2)})^{n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & (\overline{\omega}^{J(\frac{n}{2}-1)}) & (\overline{\omega}^{J(\frac{n}{2}-1)})^2 & \cdots & (\overline{\omega}^{J(\frac{n}{2}-1)})^{n-1} \end{bmatrix}$$

, and extract only the first $\frac{n}{2}$ elements in the *forward-ordered* Hermitian vector $\vec{v}$ to recover the input vector $\vec{v}$.

CKKS's Approximation Property: In the encoding process, when we convert $\vec{v} \rightarrow \vec{m} \rightarrow \Delta\vec{m}$, we multiply $\vec{v}$ by $\tilde{W}$ which contains real numbers with infinite decimals (e.g., $\sqrt{2}$) coming from Euler's formula, which we should round to the nearest integer by computing $\lceil \Delta m \rfloor$ (which we will denote as Δm throughout this section for simplicity) and thus we lose some precision. This implies that if we later decode $\Delta\vec{m}$ into $\vec{v}_d$, this value would be slightly different from the original input vector $\vec{v}$. As CKKS's encoding scheme is subject to such a small rounding error, the decryption does not perfectly match the original input vector. Such errors also propagate across homomorphic computations, because those computations are done based on approximately encoded plaintext $\lceil \Delta\vec{m} \rfloor$. As these errors are caused by throwing away the infinitely long decimal digits, they can be corrected during the decoding process only if we use an infinitely big scaling factor Δ, which is impossible because Δm_i should not overflow the ciphertext modulus q_0 of the lowest multiplicative level. Due to this limitation, CKKS is considered an *approximate* homomorphic encryption.

D-3.1.1 Example

Suppose our input complex vector's dimension $\frac{n}{2} = 2$, the bounding polynomial degree $n = 4$, and the scaling factor $\Delta = 1024$.

Our basis of the n-dimensional vector space

$$\tilde{W} = \begin{bmatrix} 1 & 1 & 1 & 1 \\ \omega^{J(1)} & \omega^{J(0)} & \overline{\omega}^{J(1)} & \overline{\omega}^{J(0)} \\ (\omega^{J(1)})^2 & (\omega^{J(0)})^2 & (\overline{\omega}^{J(1)})^2 & (\overline{\omega}^{J(0)})^2 \\ (\omega^{J(1)})^3 & (\omega^{J(0)})^3 & (\overline{\omega}^{J(1)})^3 & (\overline{\omega}^{J(0)})^3 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & 1 \\ \omega^5 & \omega & \overline{\omega}^5 & \overline{\omega} \\ \omega^2 & \omega^2 & \overline{\omega^2} & \overline{\omega}^2 \\ \omega^7 & \omega^3 & \overline{\omega}^7 & \overline{\omega}^3 \end{bmatrix}$$

, where $\omega = e^{i\pi/n} = \cos\left(\frac{\pi}{n}\right) + i\sin\left(\frac{\pi}{n}\right)$

Given this setup, suppose we have the input complex vector $\vec{v} = (1.1+4.3i,\ 3.5-1.4i)$ to encode.

First, construct the forward-ordered Hermitian vector $\vec{v}_r = (1.1+4.3i,\ 3.5-1.4i,\ 1.1-4.3i,\ 3.5+1.4i)$.

Next, convert the complex vector $\vec{v}_r$ into a real number vector $\vec{m}$ by applying the transformation:

$$\vec{m} = \frac{\tilde{W} \cdot I_n^R \cdot \vec{v}_r}{n} = \frac{1}{4} \cdot \begin{bmatrix} 1 & 1 & 1 & 1 \\ \omega^5 & \omega & \overline{\omega}^5 & \overline{\omega} \\ \omega^2 & \omega^2 & \overline{\omega}^2 & \overline{\omega}^2 \\ \omega^7 & \omega^3 & \overline{\omega}^7 & \overline{\omega}^3 \end{bmatrix} \cdot \begin{bmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix} \cdot \begin{bmatrix} 1.1+4.3i \\ 3.5-1.4i \\ 1.1-4.3i \\ 3.5+1.4i \end{bmatrix}$$

$$= \frac{W \cdot I_n^R \cdot \vec{v}_r}{n} = \frac{1}{4} \cdot \begin{bmatrix} 1 & 1 & 1 & 1 \\ \overline{\omega} & \overline{\omega}^5 & \omega & \omega^5 \\ \overline{\omega}^2 & \overline{\omega}^2 & \omega^2 & \omega^2 \\ \overline{\omega}^3 & \overline{\omega}^7 & \omega^3 & \omega^7 \end{bmatrix} \cdot \begin{bmatrix} 1.1+4.3i \\ 3.5-1.4i \\ 1.1-4.3i \\ 3.5+1.4i \end{bmatrix}$$

$$= \frac{1}{4} \cdot \begin{bmatrix} (1.1+4.3i) + (3.5-1.4i) + (1.1-4.3i) + (3.5+1.4i) \\ (1.1+4.3i)\overline{\omega} + (3.5-1.4i)\overline{\omega}^5 + (1.1-4.3i)\omega + (3.5+1.4i)\omega^5 \\ (1.1+4.3i)\overline{\omega^2} + (3.5-1.4i)\overline{\omega^2} + (1.1-4.3i)\omega^2 + (3.5+1.4i)\omega^2 \\ (1.1+4.3i)\overline{\omega}^3 + (3.5-1.4i)\overline{\omega}^7 + (1.1-4.3i)\omega^3 + (3.5+1.4i)\omega^7 \end{bmatrix}$$

$$= \frac{1}{4} \cdot \begin{bmatrix} (1.1+4.3i) + (3.5-1.4i) + (1.1-4.3i) + (3.5+1.4i) \\ (1.1+4.3i) + (3.5-1.4i)\overline{\omega}^5 + (1.1-4.3i)\omega + (3.5+1.4i)\omega^5 \\ (1.1+4.3i)\overline{\omega^2} + (3.5-1.4i)\overline{\omega^2} + (1.1-4.3i)\omega^2 + (3.5+1.4i)\omega^2 \\ (1.1+4.3i)\overline{\omega}^3 + (3.5-1.4i)\overline{\omega}^7 + (1.1-4.3i)\omega^3 + (3.5+1.4i)\omega^7 \end{bmatrix}$$

$$= \frac{1}{4} \cdot \begin{bmatrix} 9.2 \\ 1.1(\overline{\omega} + \omega) + 4.3i(\overline{\omega} - \omega) + 3.5(\overline{\omega}^5 + \omega^5) - 1.4i(\overline{\omega}^5 - \omega^5) \\ 1.1(\overline{\omega}^2 + \omega^2) + 4.3i(\overline{\omega}^2 - \omega^2) + 3.5(\overline{\omega}^2 + \omega^2) - 1.4i(\overline{\omega}^2 - \omega^2) \\ 1.1(\overline{\omega}^3 + \overline{\omega}^3) + 4.3i(\overline{\omega}^3 - \overline{\omega}^3) + 3.5(\overline{\omega}^7 + \overline{\omega}^7) - 1.4i(\overline{\omega}^7 - \overline{\omega}^7) \end{bmatrix}$$

$$= \frac{1}{4} \cdot \begin{bmatrix} 9.2 \\ 1.1\left(2\cos\frac{\pi}{4}\right) - 4.3i\left(2i\sin\frac{\pi}{4}\right) + 3.5\left(2\cos\frac{5\pi}{4}\right) + 1.4i\left(2i\sin\frac{5\pi}{4}\right) \\ 1.1\left(2\cos\frac{\pi}{2}\right) - 4.3i\left(2i\sin\frac{\pi}{2}\right) + 3.5\left(2\cos\frac{\pi}{2}\right) + 1.4i\left(2i\sin\frac{\pi}{2}\right) \\ 1.1\left(2\cos\frac{3\pi}{4}\right) - 4.3i\left(2i\sin\frac{3\pi}{4}\right) + 3.5\left(2\cos\frac{7\pi}{4}\right) + 1.4i\left(2i\sin\frac{7\pi}{4}\right) \end{bmatrix}$$

$$= \frac{1}{4} \cdot \begin{bmatrix} 9.2 \\ 1.1\left(2\frac{\sqrt{2}}{2}\right) + 4.3\left(2\frac{\sqrt{2}}{2}\right) + 3.5\left(-2\frac{\sqrt{2}}{2}\right) - 1.4\left(-2\frac{\sqrt{2}}{2}\right) \\ 1.1(2\cdot 0) + 4.3(2\cdot 1) + 3.5(2\cdot 0) - 1.4(2\cdot 1) \\ 1.1\left(2 - \frac{\sqrt{2}}{2}\right) + 4.3\left(2\frac{\sqrt{2}}{2}\right) + 3.5\left(2\frac{\sqrt{2}}{2}\right) - 1.4\left(-2\frac{\sqrt{2}}{2}\right) \end{bmatrix}$$

$$= 0.25 \cdot \begin{bmatrix} 9.2 \\ 1.1\sqrt{2} + 4.3\sqrt{2} - 3.5\sqrt{2} + 1.4\sqrt{2} \\ 1.1(0) + 4.3(2) - 3.5(0) - 1.4(2) \\ -1.1\sqrt{2} + 4.3\sqrt{2} + 3.5\sqrt{2} + 1.4\sqrt{2} \end{bmatrix} = \begin{bmatrix} 2.3 \\ 0.825\sqrt{2} \\ 1.45 \\ 2.025\sqrt{2} \end{bmatrix} \approx (2.3,\ 1.1657,\ 1.45,\ 2.8638)$$

Convert the real number vector $\vec{m}$ into a scaled integer vector $\Delta\vec{m}$ by Δ-scaling and rounding as follows:

$\Delta\vec{m} \approx \lceil \Delta\vec{m} \rfloor = \lceil 1024 \cdot (2.3,\ 1.1657,\ 1.45,\ 2.8638) \rfloor = (2355,\ 1195,\ 1485,\ 2933)$

Finally, $\vec{v} = (1.1 + 4.3i,\ 3.5 - 1.4i)$ has been encoded into the plaintext polynomial $M(X)$ as follows:

$\Delta M(X) = 2355 + 1195X + 1485X^2 + 2933X^3 \in \mathcal{R}_{\langle 4 \rangle}$

To decode $\vec{m}$, we compute:

$$\vec{v'} = \frac{W^T \cdot \Delta\vec{m}}{\Delta} = \begin{bmatrix} 1, \omega, \omega^2, \omega^3 \\ 1, \omega^3, \omega^6, \omega \\ 1, \overline{\omega}, \overline{\omega^2}, \overline{\omega^3} \\ 1, \overline{\omega^3}, \overline{\omega^6}, \overline{\omega} \end{bmatrix} \cdot \begin{bmatrix} 2355 \\ 1195 \\ 1485 \\ 2933 \end{bmatrix} \cdot \frac{1}{1024}$$

$$= \begin{bmatrix} 1, \frac{\sqrt{2}}{2} + \frac{i\sqrt{2}}{2}, i, -\frac{\sqrt{2}}{2} + \frac{i\sqrt{2}}{2} \\ 1, -\frac{\sqrt{2}}{2} + \frac{i\sqrt{2}}{2}, -i, \frac{\sqrt{2}}{2} + \frac{i\sqrt{2}}{2} \\ 1, \frac{\sqrt{2}}{2} - \frac{i\sqrt{2}}{2}, -i, -\frac{\sqrt{2}}{2} - \frac{i\sqrt{2}}{2} \\ 1, -\frac{\sqrt{2}}{2} - \frac{i\sqrt{2}}{2}, i, \frac{\sqrt{2}}{2} - \frac{i\sqrt{2}}{2} \end{bmatrix} \cdot \begin{bmatrix} 2.2998046875 \\ 1.1669921875 \\ 1.4501953125 \\ 2.8642578125 \end{bmatrix}$$

$\approx (1.0997 + 4.3007i,\ 3.5000 - 1.4003i,\ 1.0997 - 4.3007i,\ 3.5000 + 1.4003i)$

Extract the first $\frac{n}{2} = 2$ elements in the Hermitian vector $\vec{v'}$ to recover the input vector:

$(1.0997 + 4.3007i,\ 3.5000 - 1.4003i)$

$\approx (1.1 + 4.3i,\ 3.5 - 1.4i) = \vec{v}$ # The original input vector

Because of the rounding drifts for converting square roots into integers, the decoded value is slightly different from the original input complex values. This is why CKKS is called an approximate homomorphic encryption.

Source Code: Examples of CKKS encoding can be executed by running this Python script.

D-3.2 Encryption and Decryption

CKKS's encryption and decryption schemes are similar to BFV's encryption and decryption schemes (Summary ?? in ??).

⟨Summary ??⟩ CKKS Encryption and Decryption

Initial Setup:

Δ is a plaintext scaling factor for polynomial encoding, $S \xleftarrow{\$} \mathcal{R}_{\langle n,2\rangle}$. The coefficients of the polynomial S can be either binary (i.e., $\{0,1\}$) or ternary (i.e., $\{-1,0,1\}$).

Encryption Input: $\Delta M \in \mathcal{R}_{\langle n,q\rangle}, A_i \xleftarrow{\$} \mathcal{R}_{\langle n,q\rangle}, E \xleftarrow{\xi_\sigma} \mathcal{R}_{\langle n,q\rangle}$

1. Compute $B = -A \cdot S + \Delta M + E \ \in \mathcal{R}_{\langle n,q\rangle}$
2. $\mathsf{RLWE}_{S,\sigma}(\Delta M) = (A, B) \ \in \mathcal{R}^2_{\langle n,q\rangle}$

Decryption Input: $\mathsf{ct} = (A, B) \ \in \mathcal{R}^2_{\langle n,q\rangle}$

$$\mathsf{RLWE}^{-1}_{S,\sigma}(\mathsf{ct}) = \left\lceil \frac{B + A \cdot S \bmod q}{\Delta} \right\rfloor_{\frac{1}{\Delta}} = \left\lceil \frac{\Delta M + E}{\Delta} \right\rfloor_{\frac{1}{\Delta}} \approx M$$

$\lceil x \rfloor_k$ means rounding x to the nearest multiple of k

Property of Approximate Decryption:

- Unlike BFV, CKKS's each plaintext value m_i is originally not in a modulus ring, but a real number with infinite decimal digits. Therefore, it's not possible to exactly decrypt the ciphertext to the same original value.
- If each coefficient of the noise E is smaller than $\frac{\Delta}{2}$, then the decryption ensures the precision level with the multiple of $\frac{1}{\Delta}$.

D-3.3 Ciphertext-to-Ciphertext Addition

CKKS's ciphertext-to-ciphertext addition scheme is exactly the same as BFV's ciphertext-to-ciphertext addition scheme (Summary ?? in ??).

⟨Summary ??⟩ CKKS Ciphertext-to-Ciphertext Addition

$$\begin{aligned}
&\mathsf{RLWE}_{S,\sigma}(\Delta M^{\langle 1\rangle}) + \mathsf{RLWE}_{S,\sigma}(\Delta M^{\langle 2\rangle}) \\
&= (A^{\langle 1\rangle},\ B^{\langle 1\rangle}) + (A^{\langle 2\rangle},\ B^{\langle 2\rangle}) \\
&= (A^{\langle 1\rangle} + A^{\langle 2\rangle},\ B^{\langle 1\rangle} + B^{\langle 2\rangle}) \\
&= \mathsf{RLWE}_{S,\sigma}(\Delta(M^{\langle 1\rangle} + M^{\langle 2\rangle}))
\end{aligned}$$

D-3.4 Ciphertext-to-Plaintext Addition

CKKS's ciphertext-to-plaintext addition scheme is exactly the same as BFV's ciphertext-to-plaintext addition scheme (Summary ?? in ??).

⟨Summary ??⟩ CKKS Ciphertext-to-Plaintext Addition

$\mathsf{RLWE}_{S,\sigma}(\Delta M) + \Delta\Lambda$
$= (A,\ B) + \Delta\Lambda$
$= (A,\ B + \Delta \cdot \Lambda)$
$= \mathsf{RLWE}_{S,\sigma}(\Delta(M + \Lambda))$

D-3.5 Homomorphic Ciphertext-to-Ciphertext Multiplication

CKKS's ciphertext-to-ciphertext multiplication is partially different from that of BFV. In the case of BFV, its ciphertext modulus stays the same after each multiplication. On the other hand, CKKS reduces its ciphertext modulus size by 1 after each multiplication (which is equivalent to reducing its multiplicative level by 1). When the level reaches 0, no more multiplication can be further done (unless we bootstrap the modulus). This difference happens because the two schemes use different strategies in handling their plaintext scaling factors– BFV's $\Delta = \left\lfloor \frac{q}{t} \right\rfloor$, whereas CKKS's Δ can be any value such that $\Delta \ll q_0$ where q_0 is the lowest multiplicative level's ciphertext modulus. However, both schemes use the similar relinearization technique.

To make it easy to understand, we will explain CKKS's ciphertext-to-ciphertext multiplication based on this alternate version of RLWE (Theorem **??** in **??**), where the sign of the AS term is flipped in the encryption and decryption formulas.

Suppose we have the following two (CKKS) RLWE ciphertexts:
$\mathsf{RLWE}_{S,\sigma}(\Delta M^{\langle 1 \rangle}) = (A^{\langle 1 \rangle}, B^{\langle 1 \rangle})$, where $B^{\langle 1 \rangle} = -A^{\langle 1 \rangle} \cdot S + \Delta M^{\langle 1 \rangle} + E^{\langle 1 \rangle}$
$\mathsf{RLWE}_{S,\sigma}(\Delta M^{\langle 2 \rangle}) = (A^{\langle 2 \rangle}, B^{\langle 2 \rangle})$, where $B^{\langle 2 \rangle} = -A^{\langle 2 \rangle} \cdot S + \Delta M^{\langle 2 \rangle} + E^{\langle 2 \rangle}$

RLWE ciphertext-to-ciphertext multiplication is comprised of the following 2 steps:

1. Find a formula for the *synthetic* ciphertext that is equivalent to $\mathsf{RLWE}_{S,\sigma}(\Delta^2 \cdot M^{\langle 1 \rangle} \cdot M^{\langle 2 \rangle})$ by leveraging the following congruence relation:
   $\mathsf{RLWE}_{S,\sigma}(\Delta^2 \cdot M^{\langle 1 \rangle} \cdot M^{\langle 2 \rangle}) = \mathsf{RLWE}_{S,\sigma}(\Delta \cdot M^{\langle 1 \rangle}) \cdot \mathsf{RLWE}_{S,\sigma}(\Delta \cdot M^{\langle 2 \rangle})$

2. Rescale $\mathsf{RLWE}_{S,\sigma}(\Delta^2 \cdot M^{\langle 1 \rangle} \cdot M^{\langle 2 \rangle})$ to $\mathsf{RLWE}_{S,\sigma}(\Delta \cdot M^{\langle 1 \rangle} \cdot M^{\langle 2 \rangle})$.

We will explain each of these steps.

D-3.5.1 Synthetic Ciphertext Derivation

The 1st step of RLWE ciphertext-ciphertext multiplication is to find a way to express the following congruence relation:
$\mathsf{RLWE}_{S,\sigma}(\Delta^2 \cdot M^{\langle 1 \rangle} \cdot M^{\langle 2 \rangle}) = \mathsf{RLWE}_{S,\sigma}(\Delta \cdot M^{\langle 1 \rangle}) \cdot \mathsf{RLWE}_{S,\sigma}(\Delta \cdot M^{\langle 2 \rangle})$

in terms of our following known values: $A^{\langle 1 \rangle},\ B^{\langle 1 \rangle},\ A^{\langle 2 \rangle},\ B^{\langle 2 \rangle},\ S$. First, notice that the following is true:

$\mathsf{RLWE}^{-1}_{S,\sigma}(\ \mathsf{RLWE}_{S,\sigma}(\Delta^2 \cdot M^{\langle 1 \rangle} \cdot M^{\langle 2 \rangle})\ ) = \mathsf{RLWE}^{-1}_{S,\sigma}(\ \mathsf{RLWE}_{S,\sigma}(\Delta \cdot M^{\langle 1 \rangle})\ ) \cdot \mathsf{RLWE}^{-1}_{S,\sigma}(\ \mathsf{RLWE}_{S,\sigma}(\Delta \cdot M^{\langle 2 \rangle})\ )$

, because encrypting and decrypting the multiplication of two plaintexts should give the same result as decrypting two encrypted plaintexts and then multiplying them. As the encryption and decryption functions cancel out, we get the following:
$\Delta^2 \cdot M^{\langle 1 \rangle} \cdot M^{\langle 2 \rangle} \approx (\Delta \cdot M^{\langle 1 \rangle} + E^{\langle 1 \rangle}) \cdot (\Delta \cdot M^{\langle 2 \rangle} + E^{\langle 2 \rangle})$
$= \mathsf{RLWE}^{-1}_{S,\sigma}(\ \mathsf{RLWE}_{S,\sigma}(\Delta \cdot M^{\langle 1 \rangle})\ ) \cdot \mathsf{RLWE}^{-1}_{S,\sigma}(\ \mathsf{RLWE}_{S,\sigma}(\Delta \cdot M^{\langle 2 \rangle})\ )$
where $(\Delta \cdot M^{\langle 1 \rangle} + E^{\langle 1 \rangle}) \cdot (\Delta \cdot M^{\langle 2 \rangle} + E^{\langle 2 \rangle}) = \Delta^2 \cdot M^{\langle 1 \rangle} \cdot M^{\langle 2 \rangle} + \Delta \cdot M^{\langle 1 \rangle} \cdot E^{\langle 2 \rangle} + \Delta \cdot M^{\langle 2 \rangle} \cdot E^{\langle 1 \rangle} + E^{\langle 1 \rangle} \cdot E^{\langle 2 \rangle}$, where $E^{\langle 1 \rangle} \cdot E^{\langle 2 \rangle}$ is small enough to be eliminated upon decryption, and $\Delta \cdot M^{\langle 1 \rangle} \cdot E^{\langle 2 \rangle}$ and $\Delta \cdot M^{\langle 2 \rangle} \cdot E^{\langle 1 \rangle}$ will be scaled down to $M^{\langle 1 \rangle} \cdot E^{\langle 2 \rangle}$ and $M^{\langle 2 \rangle} \cdot E^{\langle 1 \rangle}$ upon modulus switch later, becoming small enough to be enough to be eliminated during decryption

Remember from **??** the following:
$\mathsf{RLWE}^{-1}_{S,\sigma}(\ C = (A, B)\ ) = \Delta M + E = B + A \cdot S$

Thus, the above congruence relation can be rewritten as follows:

$\Delta^2 \cdot M^{\langle 1 \rangle} \cdot M^{\langle 2 \rangle} \approx (\Delta \cdot M^{\langle 1 \rangle} + E^{\langle 1 \rangle}) \cdot (\Delta \cdot M^{\langle 2 \rangle} + E^{\langle 2 \rangle})$
$= (B^{\langle 1 \rangle} + A^{\langle 1 \rangle} \cdot S - E^{\langle 1 \rangle}) \cdot (B^{\langle 2 \rangle} + A^{\langle 2 \rangle} \cdot S - E^{\langle 2 \rangle})$
$\approx (B^{\langle 1 \rangle} + A^{\langle 1 \rangle} \cdot S) \cdot (B^{\langle 2 \rangle} + A^{\langle 2 \rangle} \cdot S)$
$= B^{\langle 1 \rangle} B^{\langle 2 \rangle} + (B^{\langle 2 \rangle} A^{\langle 1 \rangle} + B^{\langle 1 \rangle} A^{\langle 2 \rangle}) \cdot S + (A^{\langle 1 \rangle} S) \cdot (A^{\langle 2 \rangle} S)$

$= \underbrace{B^{\langle 1 \rangle} B^{\langle 2 \rangle}}_{D_0} + \underbrace{(B^{\langle 2 \rangle} A^{\langle 1 \rangle} + B^{\langle 1 \rangle} A^{\langle 2 \rangle})}_{D_1} \cdot S + \underbrace{(A^{\langle 1 \rangle} \cdot A^{\langle 2 \rangle})}_{D_2} \cdot \underbrace{(S \cdot S)}_{S^2}$
$= D_0 + D_1 \cdot S + D_2 \cdot S^2$

$= \mathsf{RLWE}^{-1}_{S,\sigma}(\ C_\alpha = (D_1, D_0)\ ) + D_2 \cdot S^2$ # since $D_0 + D_1 \cdot S = \mathsf{RLWE}^{-1}_{S,\sigma}(\ C_\alpha = (D_1, D_0)\ )$

In the final step above, we converted $D_0 + D_1 \cdot S$ into $\mathsf{RLWE}^{-1}_{S,\sigma}(\ C_\alpha = (D_1, D_0)\ )$, where C_α is the synthetic RLWE ciphertext (D_1, D_0) encrypted by S. Similarly, our next task is to derive a synthetic RLWE ciphertext C_β such that $D_2 \cdot S^2 = \mathsf{RLWE}^{-1}_{S,\sigma}(C_\beta)$. The reason why we want this synthetic ciphertext is that we do not want the square of S (i.e., S^2), because if we continue to keep S^2, then over more consequent ciphertext-to-ciphertext multiplications, this term will aggregate exponentially growing bigger exponents such as $S^4, S^8, \cdots$ ..., which would exponentially increase the computational overhead of decryption. In the next subsection, we will explain how to derive the synthetic RLWE ciphertext C_β such that $D_2 \cdot S^2 = \mathsf{RLWE}^{-1}_{S,\sigma}(C_\beta)$.

D-3.5.2 Relinearization Method 1 – Ciphertext Decomposition

As explained in BFV's ciphertext-to-ciphertext multiplication (**??**), relinearization is a process of converting the polynomial triplet $(D_0, D_1, D_2) \in \mathcal{R}^3_{\langle n,q \rangle}$, which can be decrypted into ΔM using S and S^2 as keys, into the polynomial pairs $(C_\alpha, C_\beta) \in \mathcal{R}^2_{\langle n,q \rangle}$, which can be decrypted into the same ΔM by using S as key. In the previous subsection, we explained that we can convert D_0 and D_1 into C_α simply by viewing D_0 and D_1 as $C_\alpha = (D_1, D_0)$. The process of converting D_2 into C_β is

exactly the same as the technique explained in **??**, which applies the gadget decomposition (**??**) on D_2 and computes an inner product with the RLev encryption (**??**) of S^2. Specifically, we compute the following:

$\mathsf{RLWE}^{-1}_{S,\sigma}(C_\beta = \langle \mathsf{Decomp}^{\beta,l}(D_2),\ \mathsf{RLev}^{\beta,l}_{S,\sigma}(S^2)\rangle)$ # the scaling factors of $\mathsf{RLev}^{\beta,l}_{S,\sigma}(S^2)$ are all 1

$= D_{2,1}(E'_1 + S^2\frac{q}{\beta}) + D_{2,2}(E'_2 + S^2\frac{q}{\beta^2}) + \cdots + D_{2,l}(E'_l + S^2\frac{q}{\beta^l})$

$= \sum_{i=1}^{l}(E'_i \cdot D_{2,i}) + S^2 \cdot (D_{2,1}\frac{q}{\beta} + D_{2,2}\frac{q}{\beta^2} + \cdots + D_{2,l}\frac{q}{\beta^l})$

$= \sum_{i=1}^{l}\epsilon_i + D_2 \cdot S^2$ # where $\epsilon_i = E'_i \cdot D_{2,i}$

$\approx D_2 \cdot S^2$ # because $\sum_{i=1}^{l}\epsilon_i \ll D_2 \cdot E''$ (where E'' is the noise embedded in $\mathsf{RLWE}_{S,\sigma}(S^2)$

Finally, we get the following relation:

$\mathsf{RLWE}_{S,\sigma}(\Delta^2 \cdot M^{\langle 1\rangle} \cdot M^{\langle 2\rangle}) \approx C_\alpha + C_\beta$, where $C_\alpha = (D_1, D_0),\ C_\beta = \langle \mathsf{Decomp}^{\beta,l}(D_2),\ \mathsf{RLev}^{\beta,l}_{S,\sigma}(S^2)\rangle$

In the next subsection, we introduce another (older) relinearization technique.

D-3.5.3 Relinearization Method 2 – Ciphertext Modulus Switch

At the setup stage of the RLWE scheme, we craft a special pair of polynomials modulo q as follows:

$A' \xleftarrow{\$} \mathcal{R}^k_{\langle n,q\rangle}$

$E' \xleftarrow{\sigma} \mathcal{R}_{\langle n,q\rangle}$

$evk = (A',\ -A' \cdot S + E' + S^2) \in \mathcal{R}^2_{\langle n,q\rangle}$

evk is called an evaluation key, which is essentially a RLWE ciphertext of S^2 encrypted by the secret key S without any scaling factor Δ. Remember that our goal is to find a synthetic RLWE ciphertext C_β such that decrypting it gives us $D_2 \cdot S^2$, that is: $\mathsf{RLWE}^{-1}_{S,\sigma}(C_\beta) = D_2 \cdot S^2$. Let's suppose that $C_\beta = D_2 \cdot evk$. Then, decrypting C_β gives us the following:

$\mathsf{RLWE}^{-1}_{S,\sigma}(C_\beta = (D_2 \cdot evk)) = \mathsf{RLWE}^{-1}_{S,\sigma}(\ C = (D_2A',\ -D_2A' \cdot S + D_2E' + D_2 \cdot S^2)\ )$

$= D_2A' \cdot S - D_2A' \cdot S + D_2E' + D_2 \cdot S^2$

$= D_2E' + D_2 \cdot S^2$

But unfortunately, $D_2E' + D_2 \cdot S^2 \not\approx D_2 \cdot S^2$, because $D_2E' \not\approx 0$ (as D_2 is not necessarily a small number).This is because $D_2 = A^{\langle 1\rangle} \cdot A^{\langle 2\rangle}$, D_2E' can be any arbitrary value between $[0, q)$.

To solve the above problem, we modify the evaluation key as a set of polynomials in big modulo g as follows:

$A' \xleftarrow{\$} \mathcal{R}^k_{\langle n,q\rangle}$

$E' \xleftarrow{\sigma} \mathcal{R}_{\langle n,q\rangle}$

$g \xleftarrow{\$} \mathbb{Z}_{q_L^2}$ # where g is some large integer power of 2, q_L is the largest modulo before any relinearization

$evk_g = (A', -A' \cdot S + E' + gS^2) \in \mathcal{R}^2_{\langle n,gq\rangle}$

evk_g is essentially an RLWE ciphertext of gS^2 encrypted by S. We can derive the following:

$evk_g = (A', -A' \cdot S + E' + gS^2) \in \mathcal{R}^2_{\langle n,gq\rangle}$

$= (A' \bmod gq,\ -A' \cdot S + E' + gS^2 \bmod gq)$

$= (A' + k_2gq, \ -A' \cdot S + E' + gS^2 + k_1gq)$ (for some integers k_1, k_2)

Note that $D_2 = A^{\langle 1 \rangle} \cdot A^{\langle 2 \rangle} \in \mathcal{R}_{\langle n,q \rangle}$
$= D_2 \bmod q$
$= D_2 + k_3q$ (for some integer k_3)

Now, let's multiply D_2 to each component of evk_g as follows:
$(A' + k_2gq, \ -A' \cdot S + E' + gS^2 + k_1gq) \cdot (D_2 + k_3q)$
$= (\ D_2A' + D_2k_2gq + k_3qA' + k_3qk_2gq,$
$-D_2A' \cdot S + D_2E' + gD_2 \cdot S^2 + D_2k_1gq - k_3qA' \cdot S + k_3qE' + k_3qgS^2 + k_3qk_1gq)$

Now, we switch the modulus of this RLWE ciphertext from $gq \to q$ based on the technique in **??**:

$$\left(\left\lceil \frac{D_2A'}{g} \right\rfloor + \left\lceil \frac{D_2k_2gq}{g} \right\rfloor + \left\lceil \frac{k_3qA'}{g} \right\rfloor + \left\lceil \frac{k_3qk_2gq}{g} \right\rfloor, \ -\left\lceil \frac{D_2A' \cdot S}{g} \right\rfloor + \left\lceil \frac{D_2E'}{g} \right\rfloor + \left\lceil \frac{gD_2 \cdot S^2}{g} \right\rfloor + \left\lceil \frac{D_2k_1gq}{g} \right\rfloor - \left\lceil \frac{k_3qA' \cdot S}{g} \right\rfloor + \left\lceil \frac{k_3qE'}{g} \right\rfloor + \left\lceil \frac{k_3qgS^2}{g} \right\rfloor + \left\lceil \frac{k_3qk_1gq}{g} \right\rfloor\right)$$

$$= \left(\left\lceil \frac{D_2A'}{g} \right\rfloor + D_2k_2q + \left\lceil \frac{k_3qA'}{g} \right\rfloor + k_3qk_2q, \right.$$
$$\left. -\left\lceil \frac{D_2A' \cdot S}{g} \right\rfloor + \left\lceil \frac{D_2E'}{g} \right\rfloor + D_2 \cdot S^2 + D_2k_1q - \left\lceil \frac{k_3qA' \cdot S}{g} \right\rfloor + \left\lceil \frac{k_3qE'}{g} \right\rfloor + k_3qS^2 + k_3qk_1q\right)$$

$$= \left(\left\lceil \frac{D_2A'}{g} \right\rfloor + \left\lceil \frac{k_3qA'}{g} \right\rfloor \bmod q, \ -\left\lceil \frac{D_2A' \cdot S}{g} \right\rfloor + \left\lceil \frac{D_2E'}{g} \right\rfloor + D_2 \cdot S^2 - \left\lceil \frac{k_3qA' \cdot S}{g} \right\rfloor + \left\lceil \frac{k_3qE'}{g} \right\rfloor \bmod q\right)$$

$= C_\beta \in \mathcal{R}^2_{n,q}$

Now, we finally got C_β which is in the form of RLWE ciphertext modulo q. Remember that our goal is to express $D_2 \cdot S^2$ as a decryption of RLWE ciphertext. If we treat C_β as a synthetic RLWE ciphertext and decrypt it, we get the following:

$\mathsf{RLWE}^{-1}_{S,\sigma}(C_\beta)$ # where C_β is treated as a synthetic RLWE ciphertext

$$= \mathsf{RLWE}^{-1}_{S,\sigma}\left(\left(-\left\lceil \frac{D_2A' \cdot S}{g} \right\rfloor + \left\lceil \frac{D_2E'}{g} \right\rfloor + D_2 \cdot S^2 - \left\lceil \frac{k_3qA' \cdot S}{g} \right\rfloor + \left\lceil \frac{k_3qE'}{g} \right\rfloor, \ \left\lceil \frac{D_2A'}{g} \right\rfloor + \left\lceil \frac{k_3qA'}{g} \right\rfloor\right)\right)$$

$$= -\left\lceil \frac{D_2A' \cdot S}{g} \right\rfloor + \left\lceil \frac{D_2E'}{g} \right\rfloor + D_2 \cdot S^2 - \left\lceil \frac{k_3qA' \cdot S}{g} \right\rfloor + \left\lceil \frac{k_3qE'}{g} \right\rfloor + \left\lceil \frac{D_2A'}{g} \right\rfloor \cdot S + \left\lceil \frac{k_3qA'}{g} \right\rfloor \cdot S$$

$$\approx \left\lceil \frac{D_2E'}{g} \right\rfloor + D_2 \cdot S^2 + \left\lceil \frac{k_3qE'}{g} \right\rfloor \quad \# -\left\lceil \frac{D_2A' \cdot S}{g} \right\rfloor + \left\lceil \frac{D_2A'}{g} \right\rfloor \cdot S = -\left\lceil \frac{k_3qA' \cdot S}{g} \right\rfloor + \left\lceil \frac{k_3qA'}{g} \right\rfloor \cdot S \approx 0$$

$$\approx D_2 \cdot S^2 \quad \# \left\lceil \frac{D_2E'}{g} \right\rfloor \approx 0, \quad \left\lceil \frac{k_3qE'}{g} \right\rfloor \approx 0$$

As shown in the above, decrypting C_β gives us $D_2 \cdot S^2$. Therefore, we reach the following conclusion:

$$\Delta^2 \cdot M^{\langle 1 \rangle} \cdot M^{\langle 2 \rangle} \approx \mathsf{RLWE}^{-1}_{S,\sigma}(C_\alpha) + \mathsf{RLWE}^{-1}_{S,\sigma}(C_\beta) \quad \text{, where } C_\alpha = (D_1, D_0), \ C_\beta = \left\lceil \frac{D_2 \cdot evk_g}{g} \right\rfloor$$

Therefore, we finally get the following congruence relation:

$\mathsf{RLWE}_{S,\sigma}(\Delta^2 \cdot M^{\langle 1\rangle} \cdot M^{\langle 2\rangle}) \approx C_\alpha + C_\beta \quad$, where $C_\alpha = (D_1, D_0),\ C_\beta = \left\lceil \frac{D_2 \cdot evk_g}{g} \right\rfloor$

Our last step of ciphertext-to-ciphertext multiplication is to convert $\mathsf{RLWE}_{S,\sigma}(\Delta^2 \cdot M^{\langle 1\rangle} \cdot M^{\langle 2\rangle})$ into $\mathsf{RLWE}_{S,\sigma}(\Delta \cdot M^{\langle 1\rangle} \cdot M^{\langle 2\rangle})$, because if the result of ciphertext-to-ciphertext multiplication is $M^{\langle 1\rangle} \cdot M^{\langle 2\rangle} = M^{\langle 3\rangle}$, then for consistency purposes, the resulting RLWE ciphertext is supposed to be:

$\mathsf{RLWE}_{S,\sigma}(\Delta \cdot M^{\langle 1\rangle} \cdot M^{\langle 2\rangle}) = \mathsf{RLWE}_{S,\sigma}(\Delta \cdot M^{\langle 3\rangle})$

We will explain this process in the next subsection.

D-3.5.4 Rescaling

To convert $\mathsf{RLWE}_{S,\sigma}(\Delta^2 \cdot M^{\langle 1\rangle} \cdot M^{\langle 2\rangle})$ into $\mathsf{RLWE}_{S,\sigma}(\Delta \cdot M^{\langle 1\rangle} \cdot M^{\langle 2\rangle})$, we cannot simply divide the ciphertext $\mathsf{RLWE}_{S,\sigma}(\Delta^2 \cdot M^{\langle 1\rangle} \cdot M^{\langle 2\rangle})$ by Δ, because as explained in **??**, modulo arithmetic does not support direct division. Multiplying the RLWE ciphertext by Δ^{-1} (i.e., an inverse of Δ) does not work either, because the only useful property we can use for inverse multiplication is: $a \cdot a^{-1} \equiv 1$. If an inverse is multiplied to any other values other than its counterpart, the result is an arbitrary value. For example, if Δ^{-1} is multiplied to a noise (i.e., $\Delta^{-1}E$), then the result can be a very huge value. Thus, multiplying the RLWE ciphertext by Δ^{-1} does not help due to the unpredictable result of the noise term.

The safest way to convert $\mathsf{RLWE}_{S,\sigma}(\Delta^2 \cdot M^{\langle 1\rangle} \cdot M^{\langle 2\rangle})$ into $\mathsf{RLWE}_{S,\sigma}(\Delta \cdot M^{\langle 1\rangle} \cdot M^{\langle 2\rangle})$ is modulus switch (**??**), which is essentially modulo rescaling (**??**). For this to work, the RLWE setup stage should design the ciphertext domain q as $q_0 \cdot \Delta^L$, where L is denoted as the level of multiplication, and $q_0 \gg \Delta$ (which is important for the accuracy of homomorphic modulo reduction during bootstrapping in **??**). Upon each ciphertext-to-ciphertext multiplication, we switch the modulus of the RLWE ciphertext from $q_0 \cdot \Delta^i \to q_0 \cdot \Delta^{i-1}$, which effectively converts the plaintext's squared scaling factor Δ^2 (in $\mathsf{RLWE}_{S,\sigma}(\Delta^2 \cdot M^{\langle 1\rangle} \cdot M^{\langle 2\rangle})$) into Δ (in $\mathsf{RLWE}_{S,\sigma}(\Delta \cdot M^{\langle 1\rangle} \cdot M^{\langle 2\rangle})$). Once the RLWE ciphertext's level reaches 0 (i.e., ciphertext modulus q_0), we cannot do any more ciphertext-to-ciphertext multiplication, in which case we need a special process called bootstrapping to re-initialize the modulus level to L.

However, one problem with this setup is that Δ^L will be a huge number. Performing homomorphic addition or multiplication over modulo Δ^L is computationally expensive. To reduce the overhead of ciphertext size, we use the Chinese remainder theorem (**??**): given an integer x mod W where W is a multiplication of $L+1$ co-primes such that $W = w_0 w_1 w_2 w_3 \cdots w_L$, the following congruence relationships hold:

$x \equiv d_0 \bmod w_0$
$x \equiv d_1 \bmod w_1$
$x \equiv d_2 \bmod w_2$
$\vdots$
$x \equiv d_L \bmod w_L$

, where $x = \sum_{m=0}^{L} d_m y_m z_m \bmod W,\ \ y_m = \frac{W}{w_m},\ \ z_m = y_m^{-1} \bmod w_m$, and $w_0 = q_0$

In other words, x mod W can be isomorphically mapped to a vector of smaller numbers $(d_0, d_1, \cdots, d_l)$ each in modulo $w_0, w_1, \cdots, w_l$, addition/multiplication with big elements in modulo W can be done by using their encoded smaller-magnitude CRT vectors element-wise, and later decode the intended big-number result. By leveraging this property, we design the CKKS scheme's maximal ciphertext modulus as $W = \prod_{m=0}^{L} w_m$, where L is the maximum multiplicative level, $w_0 = q_0 \gg \Delta$, and all other $w_i \approx \Delta$. Then, whenever reaching from the l-th to the next lower $l-1$-th multiplicative level, we switch its modulus from $q = \prod_{m=0}^{l} w_m$ to $\hat{q} = \prod_{m=0}^{l-1} w_m$ as follows:

$(C = (A, B)) \in \mathcal{R}_{\langle n,q \rangle} \rightarrow \mathsf{RLWE}_{S,\sigma}(\hat{C} = (\hat{A}, \hat{B})) \in \mathcal{R}_{\langle n,\hat{q} \rangle}$

$q = \prod_{m=0}^{l} w_m$, # where all w_m are prime numbers, $w_0 = q_0 \gg \Delta \cdot p$ to ensure the scaled plaintext ΔM during homomorphic operations never overflows the ciphertext modulus even at the lowest multiplicative level, and all other $w_i \approx \Delta$

$\hat{q} = \frac{q}{w_l}$

$\hat{A}_i = \left\lceil \frac{\hat{q}}{q} \cdot A_i \right\rfloor = \hat{a}_{i,0} + \hat{a}_{i,1}X + \hat{a}_{i,2}X^2 + \cdots + \hat{a}_{i,n-1}X^{n-1}$, where each $\hat{a}_{i,j} = \left\lceil a_{i,j}\frac{\hat{q}}{q} \right\rfloor = \left\lceil \frac{a_{i,j}}{w_l} \right\rfloor \in \mathbb{Z}_{\hat{q}}$

$\hat{B} = \left\lceil \frac{\hat{q}}{q} \cdot B \right\rfloor = \hat{b}_0 + \hat{b}_1X + \hat{b}_2X^2 + \cdots + \hat{b}_{n-1}X^{n-1}$, where each $\hat{b}_j = \left\lceil b_j\frac{\hat{q}}{q} \right\rfloor = \left\lceil \frac{b_j}{w_l} \right\rfloor \in \mathbb{Z}_{\hat{q}}$

$\mathsf{RLWE}_{S,\sigma}(\Delta M) = (\hat{A}, \hat{B}) \in \mathcal{R}_{\langle n,\hat{q} \rangle}$

The above update of $(\{A_i\}_{i=0}^{k-1}, B)$ to $(\{\hat{A}_i\}_{i=0}^{k-1}, \hat{B})$ effectively changes Δ, E to $\hat{\Delta}, \hat{E}$ as follows:

$\hat{E} = \hat{e}_0 + \hat{e}_1X + \hat{e}_2X^2 + \cdots + \hat{e}_{n-1}X^{n-1}$, where each $\hat{e}_j = \left\lceil e_j\frac{\hat{q}}{q} \right\rfloor = \left\lceil \frac{e_j}{w_l} \right\rfloor \in \mathbb{Z}_{\hat{q}}$

$\hat{\Delta} = \left\lfloor \Delta^2\frac{\hat{q}}{q} \right\rfloor = \left\lceil \frac{\Delta^2}{w_l} \right\rfloor \approx \Delta$ # If we treat $\hat{\Delta}$ as Δ, the rounding error slightly increases the noise $\hat{E}$ to $\hat{E} + E_\Delta$, while the decryption of $(\hat{A}, \hat{B})$ outputs the same M

Note that after the rescaling, the plaintext scaling factor of $(\hat{A}, \hat{B})$ is also updated to $\hat{\Delta}$. Meanwhile, M and S stay the same as before.

After we switch the modulus of the ciphertext C from $q \rightarrow \hat{q}$ by multiplying $\frac{\hat{q}}{q}$ to A and B, the encrypted original plaintext term $\Delta^2 M^{\langle 1 \rangle} M^{\langle 2 \rangle}$ will become $\Delta^2 M^{\langle 1 \rangle} M^{\langle 2 \rangle} \cdot \frac{\hat{q}}{q} = \frac{\Delta^2 M^{\langle 1 \rangle} M^{\langle 2 \rangle}}{w_l} = (\Delta + \epsilon_\Delta) \cdot M^{\langle 1 \rangle} M^{\langle 2 \rangle}$, where $\epsilon_\Delta \approx 0$, because as explained before, we chose $\{w_i\}_{i=1}^{L}$ such that $w_i \approx \Delta$. Therefore, $(\Delta + \epsilon_\Delta) \cdot M^{\langle 1 \rangle} M^{\langle 2 \rangle} = \Delta M^{\langle 1 \rangle} M^{\langle 2 \rangle} + \epsilon_\Delta M^{\langle 1 \rangle} M^{\langle 2 \rangle}$, where $\epsilon_\Delta M^{\langle 1 \rangle} M^{\langle 2 \rangle} \approx 0$, which becomes part of the noise term of the modulus-switched (i.e., rescaled) new ciphertext $\hat{C}$.

The benefit of this design of the CRT (Chinese remainder problem)-based ciphertext modulus and rescaling is that we can isomorphically decompose the huge coefficients (bigger than 64 bits) of polynomials in ciphertexts into l-dimensional Chinese remainder vectors (Theorem **??**.2 in **??**) and perform element-wise addition or multiplication for computing coefficients over the small vector elements. This promotes computational efficiency for homomorphic addition and multiplication over

a large ciphertext modulus (although the number of addition/multiplication operations increases). This technique is called Residue Number System (RNS). When CRT is used in ciphertexts, the security regarding the ciphertext modulus depends on the smallest and the largest CRT elements.

Initial Scaling Factor Δ Upon Encryption: To support multi-level multiplicative levels (using CRT), we need to modify the generic scaling factor setup presented in Summary **??** (**??**) from $\Delta = \frac{q}{t}$ to $\Delta = w_L$.

Noise Growth: Upon each step of rescaling during ciphertext-ciphertext multiplication, the noise also gets scaled down by $\frac{1}{\Delta}$ (or by $\frac{1}{w_l}$ at multiplicative level l in the case of using CRT). Therefore, if the accumulated noise is smaller than Δ (w_l in the case of using CRT), running a single ciphertext-ciphertext multiplication operation effectively reduces the noise against growing. However, during each ciphertext-to-ciphertext multiplication, the encrypted (noisy) plaintext is $(\Delta M_1 + E_1) \cdot (\Delta M_2 + E_2) = \Delta^2 M_1 M_2 + \Delta \cdot (M_1 E_2 + M_2 E_1) + E_1 E_2$, and rescaling roughly has the effect of dividing this by Δ, which approximately gives us $\Delta M_1 M_2 + M_1 E_2 + M_2 E_1 + \frac{E_1 E_2}{\Delta}$. Because of the $(M_1 E_2 + M_2 E_1)$ term, the noise actually grows compared to before ciphertext-to-ciphertext multiplication. Therefore, ciphertext-to-ciphertext multiplication inevitably increases the noise.

D-3.5.5 Comparing BFV and CKKS Bootstrapping

CKKS bootstrapping shares several common aspects with BFV bootstrapping. CKKS's ModRaise and Homomorphic Decryption steps are equivalent to BFV's Homomorphic Decryption (without modulo-q reduction) step. BFV homomorphically multiplies polynomial A and B whose coefficients are in $\mathbb{Z}_{p^e}$ with the encrypted secret key whose ciphertext modulus is q, which generates the modulo wrap-around coefficient values $p^e K$. Similarly, CKKS coefficients are in $\mathbb{Z}_{q_0}$ with the encrypted secret key whose ciphertext modulus is q_L, which generates the modulo wrap-around coefficient values $q_0 K$. However, they use different strategies to handle their modulo wrap-around values. CKKS uses evaluation of the sine function having a period of q_0 to approximately eliminate $q_0 K$ (i.e., EvalExp). On the other hand, BFV uses digit extraction to scale down $p^e K$ by p^{e-1} and then treats the remaining small pK as part of the modulo wrap-around value of the plaintext. The requirement of the digit extraction algorithm is that the plaintext inputs should be represented as base-p values, and because of this, BFV bootstrapping includes the initial step of modulus switch from $q \to p^e$, where p^e is used as the plaintext modulus after homomorphic decryption.

Both BFV and CKKS use the same strategy for their CoeffToSlot, SlotToCoeff, and Scaling Factor Re-interpretation steps.

Multiplicative Level: A critical difference between BFV and CKKS is that in BFV, the ciphertext modulus q stays the same after ciphertext-ciphertext multiplication, and there is no restriction on the number of ciphertext-ciphertext multiplications. On the other hand, in CKKS, the ciphertext modulus changes from $q_l \to q_{l-1}$ after each multiplication, and when it reaches q_0, no more multiplication can be done, unless we reset the ciphertext modulus to q_L by using the modulus bootstrapping technique (**??**).

Limitation of Noise Handling: Although CKKS's rescaling during ciphertext-to-ciphertext multiplication reduces the magnitude of noise E by Δ, it also reduces the ciphertext modulus by the same amount, and thus the relative noise-to-ciphertext-modulus ratio does not get decreased by rescaling. In order to reduce (or reset) the noise-to-modulus ratio, we need to do bootstrapping

(??), which will be explained at the end of this section.

D-3.5.6 Summary

To put all things together, CKKS's ciphertext-to-ciphertext multiplication is summarized as follows:

⟨Summary ??⟩ CKKS's Ciphertext-to-Ciphertext Multiplication

Suppose we have the following two RLWE ciphertexts:
$\mathsf{RLWE}_{S,\sigma}(\Delta M^{\langle 1\rangle}) = (A^{\langle 1\rangle}, B^{\langle 1\rangle})$, where $B^{\langle 1\rangle} = -A^{\langle 1\rangle} \cdot S + \Delta M^{\langle 1\rangle} + E^{\langle 1\rangle}$
$\mathsf{RLWE}_{S,\sigma}(\Delta M^{\langle 2\rangle}) = (A^{\langle 2\rangle}, B^{\langle 2\rangle})$, where $B^{\langle 2\rangle} = -A^{\langle 2\rangle} \cdot S + \Delta M^{\langle 2\rangle} + E^{\langle 2\rangle}$

Multiplication between these two ciphertexts is performed as follows:

1. **Basic Multiplication**
   Compute the following:

   $D_0 = B^{\langle 1\rangle} B^{\langle 2\rangle}$
   $D_1 = B^{\langle 2\rangle} A^{\langle 1\rangle} + B^{\langle 1\rangle} A^{\langle 2\rangle}$
   $D_2 = A^{\langle 1\rangle} \cdot A^{\langle 2\rangle}$

   , where $\Delta^2 M^{\langle 1\rangle} M^{\langle 2\rangle} \approx \underbrace{B^{\langle 1\rangle} B^{\langle 2\rangle}}_{D_0} + \underbrace{(B^{\langle 2\rangle} A^{\langle 1\rangle} + B^{\langle 1\rangle} A^{\langle 2\rangle})}_{D_1} \cdot S + \underbrace{(A^{\langle 1\rangle} \cdot A^{\langle 2\rangle})}_{D_2} \cdot \underbrace{S \cdot S}_{S^2}$
   $= D_0 + D_1 \cdot S + D_2 \cdot S^2$

2. **Relinearization**
   $\mathsf{RLWE}_{S,\sigma}(\Delta^2 \cdot M^{\langle 1\rangle} \cdot M^{\langle 2\rangle}) \approx \mathsf{RLWE}_{S,\sigma}(\ (D_0 + D_1 \cdot S + D_2 \cdot S^2)\ ) \approx C_\alpha + C_\beta$

   , where $C_\alpha = (D_1, D_0)$,
   $C_\beta = \langle \mathsf{Decomp}^{\beta,l}(D_2),\ \mathsf{RLev}^{\beta,l}_{S,\sigma}(S^2)\rangle$ or $\left\lceil \frac{D_2 \cdot evk_g}{g} \right\rfloor$,
   $evk_g = (-A' \cdot S + E' + gS^2, A') \in \mathcal{R}^2_{\langle n, gq\rangle}$, $g = q_L^2$, $L:$ the maximum level

3. **Rescaling**
   Switch the relinearlized ciphertext's modulus from $q \to \hat{q}$ by updating (A, B) to $(\hat{A}, \hat{B})$ as follows:

   $(\ C = (A, B)\ ) \in \mathcal{R}_{\langle n,q\rangle} \to (\ \hat{C} = (\hat{A}, \hat{B})\ ) \in \mathcal{R}_{\langle n,\hat{q}\rangle}$
   $q = \prod_{m=0}^{l} w_m$, # where all w_m are prime numbers, $w_0 = q_0 \gg \Delta \cdot p$ to ensure the plaintext ΔM during homomorphic operations never overflows the ciphertext modulus even at the lowest multiplicative level, and all other $w_i \approx \Delta$
   $\hat{q} = \frac{q}{w_l}$

$\hat{A} = \left\lceil \frac{\hat{q}}{q} \cdot A \right\rfloor = \hat{a}_0 + \hat{a}_1 X + \hat{a}_2 X^2 + \cdots + \hat{a}_{n-1} X^{n-1}$, where each $\hat{a}_i = \left\lceil a_i \frac{\hat{q}}{q} \right\rfloor = \left\lceil \frac{a_i}{w_l} \right\rfloor \in \mathbb{Z}_{\hat{q}}$

$\hat{B} = \left\lceil \frac{\hat{q}}{q} \cdot B \right\rfloor = \hat{b}_0 + \hat{b}_1 X + \hat{b}_2 X^2 + \cdots + \hat{b}_{n-1} X^{n-1}$, where each $\hat{b}_i = \left\lceil b_i \frac{\hat{q}}{q} \right\rfloor = \left\lceil \frac{b_i}{w_l} \right\rfloor \in \mathbb{Z}_{\hat{q}}$

The above update of (A, B) to $(\hat{A}, \hat{B})$ effectively changes Δ, E to $\hat{\Delta}, \hat{E}$ as follows:

$\hat{E} = \hat{e}_0 + \hat{e}_1 X + \hat{e}_2 X^2 + \cdots + \hat{e}_{n-1} X^{n-1}$, where each $\hat{e}_i = \left\lceil e_i \frac{\hat{q}}{q} \right\rfloor = \left\lceil \frac{e_i}{w_l} \right\rfloor \in \mathbb{Z}_{\hat{q}}$

$\hat{\Delta} = \left\lceil \Delta^2 \frac{\hat{q}}{q} \right\rfloor = \left\lceil \frac{\Delta^2}{w_l} \right\rfloor \approx \Delta$ # This rounding error slightly increases the noise $\hat{E}$ to $\hat{E} + E_\Delta$, while the decryption of $(\hat{A}, \hat{B})$ outputs the same plaintext M

Note that after the rescaling, the ciphertext modulus changes from $q \to \hat{q}$, and the plaintext scaling factor of $(\hat{A}, \hat{B})$ is also updated to $\hat{\Delta}$. Meanwhile, the plaintext M and the secret key S stay the same as before.

Swapping the Order of Relinearization and Rescaling: The order of relinearization and rescaling is interchangeable. Running rescaling before relinearization reduces the size of the ciphertext modulus, and therefore the subsequent relinearization can be executed faster.

D-3.6 Ciphertext-to-Plaintext Multiplication

Remember that BFV's ciphertext-to-plaintext multiplication (**??**) is performed as follows:

$\mathsf{RLWE}_{S,\sigma}(\Delta M) \cdot \Lambda$
$= (A,\ B) \cdot \Lambda$
$= (A \cdot \Lambda,\ B \cdot \Lambda)$
$= \mathsf{RLWE}_{S,\sigma}(\Delta(M \cdot \Lambda))$

, where the plaintext polynomial Λ is not scaled by Δ. However, the above relation cannot be used in CKKS's ciphertext-to-plaintext multiplication because when CKKS encodes the input vector slots into polynomial coefficients, the encoding is computed as $\vec{m} = \frac{\tilde{W} \cdot I_n^R \cdot \vec{v}'}{n}$ (Summary **??** in **??**), where the n-th root-of-unity base $\omega = e^{i\pi/n} = \cos\left(\frac{\pi}{n}\right) + i \sin\left(\frac{\pi}{n}\right)$. Since ω is usually not an integer, the encoded polynomial Λ's coefficients are usually not integers (will usually have infinite decimal digits). Therefore, we need to follow the CKKS encoding procedure's last step (Summary **??** in **??**), which scales Λ by Δ to shift an enough number of its decimal values to the integer digits, which effectively approximates the decimal coefficients to integers with high precision. Then, the resulting encrypted plaintext becomes $\Delta M \cdot \Delta\Lambda = \Delta^2 M\Lambda$. To convert $\Delta^2 M\Lambda$ into $\Delta M\Lambda$, we need to do a rescaling operation as we did in CKKS's ciphertext-to-ciphertext multiplication's (Summary **??** in **??**) last step. Therefore, CKKS's ciphertext-to-plaintext multiplication consumes one multiplicative level (whereas BFV's ciphertext-to-plaintext multiplication does not consume any multiplicative level). CKKS's ciphertext-to-plaintext multiplication is summarized as follows:

⟨Summary ??⟩ CKKS Ciphertext-to-Plaintext Multiplication

1. **Basic Multiplication**
   $\mathsf{RLWE}_{S,\sigma}(\Delta M) \cdot \Delta\Lambda$
   $= (A,\ B) \cdot \Delta\Lambda$
   $= (A \cdot \Delta\Lambda,\ B \cdot \Delta\Lambda)$
   $= \mathsf{RLWE}_{S,\sigma}(\Delta^2(M \cdot \Lambda))$

2. **Rescaling**
   Switch the relinearlized ciphertext's modulus from $q \to \hat{q}$ as done in CKKS's ciphertext-to-ciphertext multiplication's (Summary **??** in **??**) last step.

D-3.7 ModDrop

Remember that CKKS's ciphertext decryption relation is as follows:
$\Delta M + E = A \cdot S + B \bmod q_l$
$\Delta M + E = A \cdot S + B - K \cdot q_l$ # where $K \cdot q_l$ represents a modulo reduction by q_l

ModDrop is an operation of lowering the multiplicative level of a ciphertext by sequentially throwing away its modulus's one or more prime elements $\left(\text{i.e.,} \left\{\frac{q_i}{q_{i-1}}\right\}_{i=0}^{L}\right)$ except for the last one q_0, while ensuring that the plaintext's scaling factor Δ stays the same as before. Specifically, a ModDrop operation that decreases its modulus from $q_l \to q_{l-1}$ is performed by updating the ciphertext (A, B) to a new one: $(A' = A \bmod q_{l-1},\ B' = B \bmod q_{l-1})$. After the ModDrop, the ciphertext's modulus decreases from $q_l \to q_{l-1}$, yet its decryption relation still holds the same as follows:
$A' \cdot S + B' - K \cdot q_l$
$= (A \bmod q_{l-1}) \cdot S + (B \bmod q_{l-1}) - K \cdot q_l$
$= (A - K_A \cdot q_{l-1}) \cdot S + (B - K_B \cdot q_{l-1}) - K \cdot q_l$
$= A \cdot S + B - (K_A + K_B + K\frac{q}{q_{l-1}}) \cdot q_{l-1}$ # where $\frac{q}{q_{l-1}}$ is an integer (the l-the prime element of q_L)
$= A \cdot S + B - K' \cdot q_{l-1}$ # where $K' = K_A + K_B + K\frac{q}{q_{l-1}}$ is an integer
$= A \cdot S + B \bmod q_{l-1}$
$= \Delta M + E$ # since $\Delta M + E < q_0 < q_{l-1}$

As shown above, $(A', B') \bmod q_{l-1}$ decrypts to the same $\Delta M + E$, a scaled plaintext with an error.

CKKS's ModDrop is summarized as follows:

⟨Summary ??⟩ CKKS's ModDrop

Given a CKKS ciphertext with the l-th multiplicative level $\mathsf{RLWE}_{S,\sigma}(\Delta M) = (A, B) \bmod q_l$, a ModDrop operation is as follows:
$(A', B') \bmod q_{l-1} = (A \bmod q_{l-1}, B \bmod q_{l-1})$

, after which the ciphertext's multiplicative level decreases by 1, while the plaintext's scaling factor Δ and the noise are unaffected.

D-3.7.1 Difference between Modulus Switch and ModDrop

In CKKS, both modulus switch (i.e., rescaling explained in **??**) and ModDrop lower a ciphertext's modulus from $q_l \rightarrow q_{l-1}$. However, the key difference is that rescaling also decreases the plaintext's scaling factor by the $\frac{q_l}{q_{l-1}} \approx \Delta$, whereas ModDrop does not affect the plaintext's scaling factor and the noise. Therefore, rescaling is used only during ciphertext-to-ciphertext multiplication when scaling down the plaintext's scaling factor in the intermediate ciphertext from $\Delta^2 \rightarrow \Delta$. Meanwhile, ModDrop is used to reduce the modulo computation time during an application's routine when it becomes certain that the ciphertext will not undergo any additional ciphertext-to-ciphertext multiplication (i.e., no need to further decrease the ciphertext's modulus).

D-3.8 Homomorphic Key Switching

CKKS's homomorphic key switching scheme changes an RLWE ciphertext's secret key from S to S'. This scheme is exactly the same as BFV's key switching scheme (Summary **??** in **??**).

⟨Summary ??⟩ CKKS's Key Switching

$\mathsf{RLWE}_{S',\sigma}(\Delta M) = (0, B) + \langle \mathsf{Decomp}^{\beta,l}(A),\ \mathsf{RLev}^{\beta,l}_{S',\sigma}(S) \rangle$

D-3.9 Homomorphic Rotation of Input Vector Slots

CKKS's batch encoding scheme (Summary **??** in **??**) implicitly supports homomorphic rotation of input slot vectors like that of BFV's homomorphic rotation (Summary **??** in **??**). This is because CKKS uses the same encoding and decoding matrices ($\tilde{W}$ and $\tilde{W}^*$) designed for the BFV encoding and decoding scheme that supports homomorphic rotation of input vector slots. Although the roots of the $(\mu = 2n)$-th cyclotomic polynomial $X^n + 1$ are different for the BFV and CKKS schemes (as one is designed over $X \in \mathbb{Z}_t$ and the other is over $X \in \mathbb{R}$), CKKS still can use the same $\tilde{W}$ (and $\tilde{W}^*$) matrices as BFV, because the $(\mu = 2n)$-th cyclotomic polynomial over $X \in \mathbb{Z}_t$ exhibits the same essential properties as the $(\mu = 2n)$-th cyclotomic polynomial over $X \in \mathbb{R}$ (as explained in **??**). Especially, the roots of both $(\mu = 2n)$ cyclotomic polynomials are the primitive $(\mu = 2n)$-th roots of unity having the order $2n$, and those n distinct roots are defined as $\omega^1, \omega^3, \cdots, \omega^{2n-1}$, where ω can be any root. Therefore, substituting CKKS's $(\mu = 2n)$-th roots of unity into the ω terms in BFV's encoding matrix $\tilde{W}$ (and decoding matrix $\tilde{W}^*$) preserves the same computational correctness for the encoding and decoding schemes, as well as for input vector slot rotation.

Importantly, the $\tilde{W}$ and $\tilde{W}^*$ matrices in both the BFV and CKKS schemes satisfy the exact requirement for supporting input vector slot rotation. That is, given the following relations:

- The $\vec{v} \rightarrow \vec{m}$ encoding formula: $\vec{m} = n^{-1} \cdot I_n^R \cdot \tilde{W} \cdot \vec{v}$
- The $\vec{m} \rightarrow \vec{v}$ decoding formula: $\vec{v} = \tilde{W}^* \cdot \vec{m}$
- The encoded polynomial $M(X) = \sum_{i=0}^{n-1} m_i X^i$

, updating the polynomial $M(X)$ to $M(X^{J(h)})$ results in the effect of rotating the first half of the n-dimensional input vector slots ($\vec{v} \in \mathbb{Z}_p^n$ in the case of BFV, and the forward-ordered Hermitian vector $\vec{v'} \in \hat{\mathbb{C}}^n \rightarrow \mathbb{C}^{\frac{n}{2}}$ in the case of CKKS) by h positions to the left (in a wrapping manner among them) and the second half of the slots also by h positions to the right (in a wrapping manner among them).

BFV uses CKKS's same rotation scheme described in Summary **??** (in **??**) as follows:

⟨Summary ??⟩ CKKS's Homomorphic Rotation of Input Vector Slots

Suppose we have an RLWE ciphertext and a key-switching key as follows:
$\mathsf{RLWE}_{S,\sigma}(\Delta M) = (A, B), \quad \mathsf{RLev}_{S,\sigma}^{\beta,l}(S^{J(h)})$

Then, the procedure of rotating all $\frac{n}{2}$ elements of the ciphertext's original input vector $\vec{v}$ by h positions to the left is as follows:

1. Update $A(X)$, $B(X)$ to $A(X^{J(h)})$, $B(X^{J(h)})$.
2. Perform the following key switching (**??**) from $S(X^{J(h)})$ to $S(X)$:
   $\mathsf{RLWE}_{S(X),\sigma}(\Delta M(X^{J(h)})) = (0, B(X^{J(h)})) + \langle \mathsf{Decomp}^{\beta,l}(A(X^{J(h)})),\ \mathsf{RLev}_{S(X),\sigma}^{\beta,l}(S(X^{J(h)})) \rangle$

Rotation within Half Slots: Like BFV, CKKS rotates the first half of the forward-ordered Hermitian input vector slots $\vec{v'} \in \hat{\mathbb{C}}^n$ and the second half of its slots separately in a partitioned manner. This is because the first half rows of $\tilde{W}^*$ comprise the terms $\omega^{J(h)}$ for $h = \{0, 1, \cdots, \frac{n}{2} - 1\}$ (i.e., evaluates $M(X)$ at $X = \{\omega^{J(0)}, \omega^{J(1)}, \cdots, \omega^{J(\frac{n}{2}-1)}\}$), whereas the second half rows of $\tilde{W}^*$ comprise the terms $\omega^{J_*(h)}$ (i.e., evaluates $M(X)$ at $X = \{\omega^{J_*(0)}, \omega^{J_*(1)}, \cdots, \omega^{J_*(\frac{n}{2}-1)}\}$), and the computed values of $J(h)$ and $J_*(h)$ repeat (i.e., rotate) within their own rotation group across $h = \{0, 1, \cdots, \frac{n}{2} - 1\}$. Because of this structure of $\tilde{W}$ and $\tilde{W}^*$, BFV and CKKS cannot design a wrapping rotation scheme across all n slots of the input vector homogeneously, but can instead design a wrapping rotation scheme across each group of the first-half and the second-half $\frac{n}{2}$ slots of the input vector in a partitioned manner. That being said, CKKS can meaningfully only use the first $\frac{n}{2}$ slots for homomorphic computations anyway, because the latter $\frac{n}{2}$ slots are conjugates of the first $\frac{n}{2}$ slots which cannot be chosen by the user but are deterministically configured based on the first $\frac{n}{2}$. On the other hand, in BFV, the user can choose the entire $\frac{n}{2}$ according to his/her needs, so BFV's utility of slots is full n. Therefore, the user can use BFV's first-half slots and second-half slots together to perform parallel computations.

D-3.9.1 Example

In this subsection, we will show the following 2 examples:

1. Encode an input vector $\vec{v}$ into a plaintext polynomial $M(X)$ based on our updated updated encoding & decoding matrices $\tilde{W}$ and $\tilde{W}^*$
2. Rotate all elements of the input vector $\vec{v}$ h positions to the left by updating the encoded plaintext $M(X)$ to $M(X^{J(h)})$

We will use the same example of the input vector $\vec{v}$ used in **??**: $\vec{v}^{\langle h=1\rangle} = (1.1 + 4.3i, 3.5 - 1.4i)$.
Remember that the encoded plaintext polynomial of $\vec{v}$ is as follows:
$\Delta M(X) = 2355 + 1195X + 1485X^2 + 2933X^3 \in \mathcal{R}_{\langle 4\rangle} \in \mathbb{R}[X]/X^4 + 1$
Suppose we want to rotate the input vector $\vec{v}$ by 1 position to the left as follows:
$\vec{v}^{\langle h=1\rangle} = (3.5 - 1.4i, 1.1 + 4.3i)$

Therefore, we update $\Delta M(X)$ to $\Delta M(X^{J(1)})$ as follows:
$\Delta M(X^{J(1)}) = \Delta M(X^5) = 2355 + 1195(X^5) + 1485(X^5)^2 + 2933(X^5)^3$
$= 2355 + 1195X^5 + 1485X^{10} + 2933X^{15}$
$= 2355 + 1195X \cdot (-1) + 1485X^2 \cdot (-1) \cdot (-1) + 2933X^3 \cdot (-1) \cdot (-1) \cdot (-1)$
$= 2355 - 1195X + 1485X^2 - 2933X^3$

The rotated *forward-ordered* Hermitian input vector is computed as follows:

$$\frac{\tilde{W}^* \cdot \Delta\vec{m}}{\Delta} = \begin{bmatrix} 1, \omega, \omega^2, \omega^3 \\ 1, \omega^3, \omega^6, \omega \\ 1, \overline{\omega}, \overline{\omega^2}, \overline{\omega^3} \\ 1, \overline{\omega^3}, \overline{\omega^6}, \overline{\omega} \end{bmatrix} \cdot \begin{bmatrix} 2355 \\ -1195 \\ 1485 \\ -2933 \end{bmatrix} \cdot \frac{1}{1024}$$

$$= \begin{bmatrix} 1, \frac{\sqrt{2}}{2} + \frac{i\sqrt{2}}{2}, i, -\frac{\sqrt{2}}{2} + \frac{i\sqrt{2}}{2} \\ 1, -\frac{\sqrt{2}}{2} - \frac{i\sqrt{2}}{2}, i, \frac{\sqrt{2}}{2} - \frac{i\sqrt{2}}{2} \\ 1, -\frac{\sqrt{2}}{2} + \frac{i\sqrt{2}}{2}, -i, \frac{\sqrt{2}}{2} + \frac{i\sqrt{2}}{2} \\ 1, \frac{\sqrt{2}}{2} - \frac{i\sqrt{2}}{2}, -i, -\frac{\sqrt{2}}{2} - \frac{i\sqrt{2}}{2} \end{bmatrix} \cdot \begin{bmatrix} 2.2998046875 \\ -1.1669921875 \\ 1.4501953125 \\ -2.8642578125 \end{bmatrix}$$

$\approx (3.500 - 1.4003i,\ 1.0997 + 4.3007i,\ 3.500 + 1.4003i,\ 1.0997 - 4.3007i)$

Extract the first $\frac{n}{2} = 2$ elements in the above Hermitian vector to recover the input vector:
$(3.500 - 1.4003i,\ 1.0997 + 4.3007i)$
$\approx (3.5 - 1.4i,\ 1.1 + 4.3i) = \vec{v}^{\langle h=1\rangle}$ # The original input vector $\vec{v}$ rotated by 1 position to the left

In practice, we do not directly update $\Delta M(X)$ to $\Delta M(X^{J(1)})$, because we would not have access to the plaintext polynomial $M(X)$ unless we have the secret key $S(X)$. Therefore, we instead update $\mathsf{ct} = (A(X), B(X))$ to $\mathsf{ct}^{\langle h=1\rangle} = (A(X^{J(1)}), B(X^{J(1)}))$, which is equivalent to homomorphically

rotating the encrypted input vector slots. Then, decrypting $\mathsf{ct}^{\langle h=1\rangle}$ and decoding it would output $\vec{v}^{\langle h=1\rangle}$.

Source Code: Examples of CKKS's homomorphic input vector slot rotation can be executed by running this Python script.

D-3.10 Contemplation on CKKS Encoding

Why CKKS's Encoding Uses the ($\mu = 2n$)-th Cyclotomic Polynomial: At this point, it becomes clear why the CKKS encoding and decoding scheme uses the (power-of-2)-th cyclotomic polynomial (i.e., $X^n + 1$) over $X \in \mathbb{C}$ (complex numbers) (**??**). The first reason is that CKKS's first requirement for designing a valid encoding and decoding formula for an input complex vector is to isomorphically convert it into a unique real number vector (and we scale this real number vector as an integer vector and use it as a list of coefficients for polynomial encoding, because CKKS's homomorphic encryption and decryption are supported only based on polynomials with integer coefficients). As for the decoding formula of an input complex vector, our high-level idea was to treat the encoded real number vector as coefficients of an $(n-1)$-degree polynomial and evaluate this polynomial at n distinct X coordinates, whose resulting set of n distinct Y values is guaranteed to be unique within the n-th degree polynomial ring. Based on this insight, we designed a decoding matrix (**??**) in the form of a Vandermonde matrix (**??**). Then, the encoding formula is equivalent to multiplying the input complex vector by the inverse of this decoding matrix. However, in linear algebra, not all matrices are guaranteed to have a counterpart inverse matrix. Therefore, for the guarantee of the existence of a valid encoding matrix (i.e., an inverse of the decoding matrix), we leveraged the following arithmetic property: if a Vandermonde matrix $V = Vander(x_0, x_1, \cdots, x_{n-1})$ is made of n distinct primitive ($\mu = 2n$)-th roots of unity (where n is a power of 2), then such a Vandermonde matrix is guaranteed to have an inverse (**??**) counterpart. In fact, the ($\mu = 2n$)-th roots of unity are n distinct roots of the ($\mu = 2n$)-th cyclotomic polynomial: $X^n + 1$ (**??**). Therefore, CKKS uses $X^n + 1$ as the polynomial ring of its subsequent encryption and decryption scheme (**??**) as well.

The CKKS encoding's second reason for using the ($\mu = 2n$)-th cyclotomic polynomial is to design a valid input vector slot rotation scheme (**??**). In this rotation scheme, updating the encoded polynomial $M(X)$ to $M(X^{J(h)})$ (where $J(h) = 5^h \bmod 2n$) is equivalent to updating the CKKS decoding process's each evaluation coordinate of $M(X)$ from x_i to $x_i^{J(h)}$ (where each x_i is the primitive ($\mu = 2n$)-th roots of unity), which gives the same effect as vertically rotating the encoding matrix (i.e., the inverse of the Vandermonde matrix whose roots are the primitive ($\mu = 2n$)-th roots of unity) upward by h positions. And this vertical rotation of the encoding matrix (while the input vector is fixed) gives the same effect of rotating the input vector $\vec{v}$ by h positions to the left (without modifying the encoding matrix). Therefore, the ($\mu = 2n$)-th cyclotomic polynomial $X^2 + 1$ is an ideal tool to design input vector slot rotation.

D-3.11 Homomorphic Conjugation

As explained in Summary **??** (**??**), given the $\frac{n}{2}$-dimensional input vector $\vec{v} = (v_0, v_1, \cdots, v_{\frac{n}{2}-1})$, its corresponding n-dimensional Hermitian vector is $\vec{v}' = (v_0, v_1, \cdots, v_{\frac{n}{2}-1}, \overline{v}_{\frac{n}{2}-1}, \overline{v}_{\frac{n}{2}-2}, \cdots, \overline{v}_1, \overline{v}_0)$.

To compute the conjugation of $\vec{v}$, which is essentially conjugating $\vec{v}$, we can conjugate $M(X)$ as follows:

$\overline{\vec{v}} = (\overline{v}_0, \overline{v}_1, \cdots, \overline{v}_{\frac{n}{2}-1}, v_{\frac{n}{2}-1}, \cdots, v_1, v_0)$

$= (M(\overline{\omega})), M(\overline{\omega^3}), \cdots, M(\overline{\omega^{n-1}}), M(\omega^{n-1}), \cdots, M(\omega^3), M(\omega))$ # where $\omega = e^{\frac{i\pi}{n}}$

$= (M((\omega)^{-1}), M((\omega^3)^{-1}), \cdots, M((\omega^{n-1})^{-1}), M((\overline{\omega^{n-1}})^{-1}), \cdots, M((\overline{\omega^3})^{-1}), M((\overline{\omega})^{-1}))$

since $\overline{\omega^k} = e^{\overline{\frac{ki\pi}{n}}} = e^{-\frac{ki\pi}{n}} = \omega^{-k}$ and $\omega^k = (\overline{\omega^k})^{-1}$ for $k = \{1, 3, \cdots, n-1\}$

$= \{M(X^{-1})\}$ # where $X = \{\omega, \omega^3, \cdots, \omega^{n-1}, \overline{\omega^{n-1}}, \cdots, \overline{\omega^3}, \overline{\omega}\} = \{\omega^1, \omega^3, \cdots, \omega^{2n-1}\}$

Therefore, homomorphic conjugation of the input vector is equivalent to updating the ciphertext $(A(X)), B(X))$ to $(A(X^{-1}), B(X^{-1}))$ and then key-switching it from $S(X^{-1}) \to S(X)$.

⟨Summary ??⟩ CKKS's Homomorphic Conjugation

Homomorphic conjugation of the input vector of a ciphertext is equivalent to the following:
1. Update the ciphertext $(A(X)), B(X))$ to $(A(X^{-1}), B(X^{-1}))$.
2. Key-switch $(A(X^{-1}), B(X^{-1}))$ from $S(X^{-1})$ to $S(X)$.

D-3.12 Sparsely Packing Ciphertexts

In **??**, we learned the CKKS encoding scheme which encodes an input vector with $\frac{n}{2}$ slots (i.e., $\frac{n}{2}$-dimensional input vector) into an $(n-1)$-degree polynomial. While the polynomial ring's degree n is fixed at the setup stage of CKKS as a security parameter, some applications may only need to use fewer than $\frac{n}{2}$ input vector slots. Suppose we only need to use $\frac{n'}{2}$ slots out of $\frac{n}{2}$ slots, where n' is some number that divides n. Then, the corresponding input vector and encoded polynomial get a special property as described below:

⟨Summary ??⟩ CKKS's Sparsely Packing Polynomial and Ciphertext

Suppose that an $\frac{n}{2}$-dimensional input vector gets encoded into a polynomial in $\mathbb{R}[X]/(X^n + 1)$. And suppose that n' is some number that divides n. We define polynomial $M(X) \in \mathbb{R}[X]/(X^n + 1)$ as the one which has non-zero constants at the terms whose power is a multiple of $\frac{n}{n'}$ and all other terms have zero constants (i.e., $M(X) = c_0 + c_{\frac{n}{n'}} X^{\frac{n}{n'}} + c_{\frac{2n}{n'}} X^{\frac{2n}{n'}} + \cdots + c_{n-\frac{n}{n'}} X^{n-\frac{n}{n'}}$). We can express $M(X)$ as some $M_Y(Y) \in \mathbb{R}[Y]/(Y^{n'}+1)$ where $Y = X^{\frac{n}{n'}}$ and thus $M(X) = M_Y(X^{\frac{n}{n'}})$.

Then, the following are true:
1. Every $M_Y(Y) \in \mathbb{R}[Y]/(Y^{n'} + 1)$ is isomorphically mapped to (i.e., decoded into) some $\frac{n}{2}$-dimensional input vector which comprises $\frac{n}{n'}$ repetitions of $\frac{n'}{2}$ consecutive slot values.
2. Conversely, if an $\frac{n}{2}$-dimensional input vector comprises $\frac{n}{n'}$ repetitions of the first $\frac{n'}{2}$ consecutive slot values, then the vector gets encoded into some $M_Y(Y) \in \mathbb{R}[Y]/(Y^{n'}+1)$

(i.e., some polynomial in $\mathbb{R}[X]/(X^n+1)$ that has zero constants at the terms whose degree is not a multiple of $\frac{n}{n'}$).

We could show both directions of proof: (1) the forward (decoding-direction) proof; and (2) the backward (encoding-direction) proof. However, it is sufficient to prove only either direction, because the encoding (σ^{-1}) and decoding (σ) processes are isomorphic. Among these two, we will show only the forward proof for simplicity.

D-3.12.1 Forward Proof: Decoding of Sparsely Packed Ciphertext

We will prove that for each $M_Y(Y) \in \mathbb{Z}[Y]/(Y^{n'}+1)$ (i.e., a polynomial in $\mathbb{R}[X]/(X^n+1)$ which has non-zero constants only at those terms with a power of multiple of $\frac{n}{n'}$ and zero constants in all other terms), the polynomial gets decoded into some $\frac{n}{2}$-dimensional input vector which comprises $\frac{n}{n'}$ repetitions of the first $\frac{n'}{2}$ consecutive slot values.

To decode $M(X)$ into an input vector, we need to evaluate $M(X)$ at $\frac{n}{2}$ distinct roots of X^n+1 (i.e., n distinct primitive ($\mu = 2n$)-th roots of unity), which are:

$(\ M(\omega^{J(0)}),\ M(\omega^{J(1)}), \cdots, M(\omega^{J(\frac{n}{2}-1)}))$

\# where $\omega = e^{\frac{i\pi}{n}}$, the base and generator of the primitive ($\mu = 2n$)-th roots of unity

But since $M(X) = M_Y(X^{\frac{n}{n'}})$, the above evaluation is equivalent to evaluating:

$(\ M_Y((\omega^{J(0)})^{\frac{n}{n'}}),\ M_Y((\omega^{J(1)})^{\frac{n}{n'}}), \cdots, M_Y((\omega^{J(\frac{n}{2}-1)})^{\frac{n}{n'}}))$

$= (\ M_Y((\omega^{\frac{n}{n'}})^{J(0)}),\ M_Y((\omega^{\frac{n}{n'}})^{J(1)}), \cdots, M_Y((\omega^{\frac{n}{n'}})^{J(\frac{n}{2}-1)}))$

$= (\ M_Y(\xi^{J(0)}),\ M_Y(\xi^{J(1)}), \cdots, M_Y(\xi^{J(\frac{n}{2}-1)}))$

\# where $\xi = e^{\frac{i\pi}{n'}}$, the base and generator of the primitive ($\mu = 2n'$)-th roots of unity

Notice that $\xi = \omega^{\frac{n}{n'}}$. Therefore, the above evaluation of $M_Y(Y)$ outputs $\frac{n}{n'}$ repeated values of $M_Y(Y)$ evaluated at $\frac{n'}{2}$ distinct primitive ($\mu = 2n'$)-th roots of unity.

D-3.13 Modulus Bootstrapping

- Reference: Bootstrapping for Approximate Homomorphic Encryption [**?**]

During CKKS's ciphertext-to-ciphertext multiplication, each ciphertext is associated with a particular multiplicative level and it decreases by 1 upon each ciphertext-to-ciphertext multiplication (by its internal modulus rescaling operation). Reaching multiplicative level 0 is equivalent to reaching the end of a ciphertext's modulus chain and no more ciphertext-to-ciphertext multiplication can be performed. To continue with further ciphertext-to-ciphertext multiplication, CKKS provides a special operation called *bootstrapping*, which is a process of resetting the ciphertext's end-of-chain modulus q_0 to the initial maximum modulus q_L (which is either $q_0 \cdot \Delta^L$ in the vanilla rescaling scheme, or $\prod_{m=0}^{L} w_m$ in the case of using CRT, as explained in **??**).

Suppose we have a ciphertext (A, B) with multiplicative depth 0. If we decrypt a ciphertext whose multiplicative level is 0 (i.e., the ciphertext's modulus is q_0), then decrypting it *without* reduction modulo q_0 would output:

$\mathsf{RLWE}^{-1}(\mathsf{ct} = (A, B))$

$= B + A \cdot S = \Delta M + E + q_0 \cdot K$ # since $B + A \cdot S \bmod q_0 = \Delta M + E$

, where $q_0 \cdot K$ accounts for wrap-around modulo q_0 values– each coefficient of polynomial $q_0 K$ is some multiple of q_0. CKKS's bootstrapping procedure is equivalent to *safely* transforming a ciphertext's modulus from q_0 to q_L (where $q_L \gg q_0$).

D-3.13.1 High-level Idea

As the first step of bootstrapping, we forcibly change the modulus of the ciphertext (A, B) from q_0 to q_L. Then, its decryption with reduction modulo q_L would output:

$\mathsf{RLWE}^{-1}(\mathsf{ct} = (A, B)) \bmod q_L$

$= B + A \cdot S \bmod q_L$

$= \Delta M + E + q_0 K \bmod q_L$

Here, we assume that q_L is large enough such that $\Delta M + E + q_0 K \ll q_L$. This is true given S has small coefficients which are either $\{-1, 0, 1\}$, and thus the coefficients of $B + A \cdot S$ would not grow much.

In the $\Delta M + E + q_0 K \bmod q_L$ term, notice that because of the $q_0 K$ term which is not modulo-reduced by q_0 anymore, the ciphertext's decrypted plaintext polynomial's each i-th term would get a corrupted coefficient $\Delta m_i + e_i + q_0 \cdot k_i \bmod q_L$ instead of $\Delta m_i + e_i \bmod q_L$. So, we now need to eliminate the garbage term $q_0 \cdot k_i \bmod q_L$ in each coefficient and distill the pure plaintext coefficient $\Delta m_i + e_i$.

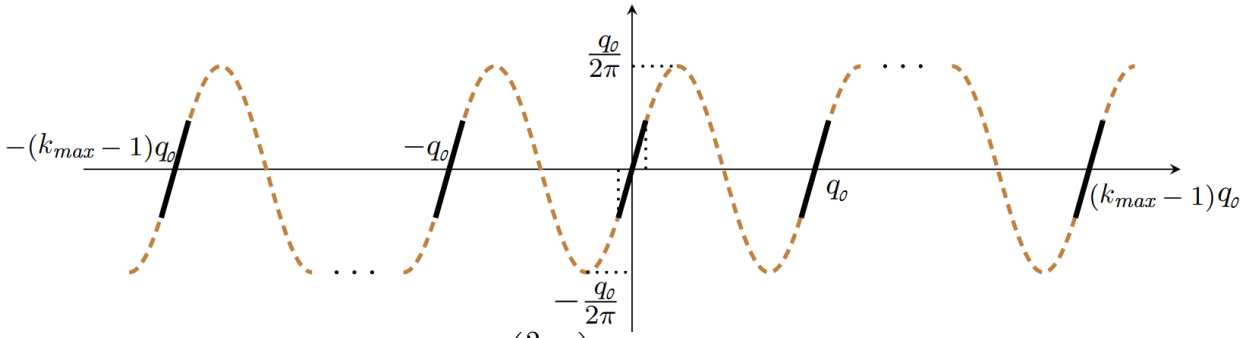


Figure 16: Sine function $f(x) = \frac{q_0}{2\pi} \cdot \sin\left(\frac{2\pi x}{q_0}\right)$ such that $f(\Delta m_i + e_i + q_0 k_i) \approx \Delta m_i + e_i$ (provided $\Delta m_i + e_i \ll q_0$) (Source)

To do so, we will take an approximated approach by using a sine function described in **??**, which has a period of q_0 with the amplitude $\frac{q_0}{2\pi}$. This sine function has the following two useful properties:

1. When $f(x)$ is evaluated at x values near the multiple of q_0, the result approximates to that of a line function $y = x$. This is because the derivative (slope) of $\sin x$ is $y' = \cos x$, and if x is a multiple of 2π, the slope is: $y' = \cos 2\pi = 1$.
2. The evaluation of $f(x)$ eliminates the multiples of q_0 from the input (i.e., modulo reduction q_0)

Combining these two properties, given input $x = \Delta m_i + e_i + q_0 k_i$,

1	**ModRaise:** Given ciphertext (A, B) mod q_0, we forcibly modify its modulus from q_0 to q_L. Then, it ends up encrypting $\Delta M + E + q_0 k$ instead of $\Delta M + E$.
2	**CoeffToSlot:** Based on step 1's ciphertext (A, B) mod q_L, we generate a new ciphertext that encrypts an input vector whose its each i-th slot stores $\Delta m_i + e_i + q_0 k_i$. This is equivalent to moving the coefficients of polynomial $\Delta M + E + q_0 K$ to the input vector slots.
3	**EvalExp:** We convert the sine function into an approximated polynomial by using the Taylor series, as well as other optimizations such as Euler's formula ($e^{i\theta} = \cos(\theta) + i \cdot \sin(\theta)$). Then, we generate a CKKS plaintext that encodes this approximated sine function, and then use this to homomorphically evaluate step 2's encrypted vector elements (to homomorphically remove every $q_0 k_i$.
4	**SlotToCoeff:** Based on the resulting ciphertext from step 3, we generate a new ciphertext whose encrypted polynomial's each i-th coefficient is (approximately) $\Delta m_i + e_i$. This is equivalent to moving the $q_0 k_i$-eliminated values stored in the input vector slots in step 3 back to the positions of the polynomial coefficients. The final ciphertext is our goal ciphertext that (approximately) encrypts $\Delta M + E$ under modulus q_L.

Table 6: High-level Description of CKKS's Bootstrapping Procedure

$$f(\Delta m_i + e_i + q_0 k_i) = \frac{q_0}{2\pi} \cdot \sin\left(\frac{2\pi \cdot (\Delta m_i + e_i + q_0 k_i))}{q_0}\right) = \frac{q_0}{2\pi} \cdot \sin\left(\frac{2\pi \cdot (\Delta m_i + e_i))}{q_0}\right) \approx \Delta m_i + e_i$$

, provided $\Delta m_i + e_i$ is very close to 0 relative to q_0 (i.e., $\Delta m_i + e_i \ll q_0$). This is true, because by construction of the CKKS scheme, the plaintext modulus (even with scaling it up by Δ), is significantly smaller than the ciphertext modulus. Therefore, to remove $q_0 K$ from $\Delta M + E + q_0 K$, we can update each coefficient of the polynomial $\Delta M + E + q_0 K$ by evaluating it with the $f(x)$ sine function. However, we cannot directly update the coefficients of the polynomial, because the CKKS scheme (the RLWE scheme in general) only supports the input vector's slot-wise $(+, \cdot)$ operations. Therefore, to update the polynomial coefficients, we need to express the update logic in terms of slot-wise input vector arithmetic $(+, \cdot)$. Considering all these, CKKS's overall bootstrapping procedure is described in **??**.

D-3.13.2 Mathematical Expansion of the High-level Idea

We will mathematically walk through how the bootstrapping procedure (**??**) correctly updates the modulus of the input ciphertext from q_0 to q_L.

For ease of understanding, we will first explain how we would do modulus bootstrapping for a ciphertext with multiplicative level 0 (i.e., its modulus is q_0) in case we have access to the secret key $S(X)$. Using this key, we can decrypt the ciphertext as follows:

$\mathsf{RLWE}^{-1}(\mathsf{ct} = (A, B))$ # where $\mathsf{ct} = (A, B) = \mathsf{RLWE}_{S,\sigma}(\Delta M)$

$= B + A \cdot S = \Delta M + E \bmod q_0$

$= \Delta M + E + q_0 K$ # where $q_0 K$ accounts for any potential wrap-around modulo q_0 values

Our initial goal is to bootstrap the modulus of the ciphertext from q_0 to q_L by using only the following three tools:

- Secret key S
- Batch-encoding (σ^{-1}) and decoding (σ) formulas
- Batched slot-wise $(+, \cdot)$ operation of input vectors based on their batch-encoded polynomials

After explaining the above, we will then explain how to achieve the same bootstrapping without having access to the secret key S.

ModRaise: This step forcibly changes the ciphertext's modulus from q_0 to q_L and then decrypts the ciphertext as follows:

$\mathsf{RLWE}^{-1}(\mathsf{ct} = (A, B)) = B + A \cdot S = \Delta M + E + q_0 K \bmod q_L$

Notice that the ciphertext's decrypted plaintext polynomial's each i-th coefficient gets corrupted to $m_i + e_i + q_0 \cdot k_i \bmod q_L$. So, we now need to eliminate the garbage term $q_0 k_i \bmod q_L$ in each coefficient and distill the pure plaintext coefficient $\Delta m_i + e_i$.

CoeffToSlot: This step generates a new plaintext polynomial whose each i-th input vector slot stores the corrupted coefficient $(m_i + e_i + q_0 k_i)$. The trick of doing this is to apply CKKS's batch-encoding mapping σ^{-1} (which represents the transformation $\vec{m} = \frac{\tilde{W} \cdot I_n^R \cdot \vec{v}}{n}$ as explained in **??**) to the input vector slots that encode the polynomial $\Delta M + E + q_0 K \bmod q_L$. Let $\vec{v}_c$ be the input vector that corresponds to polynomial $\Delta M + E + q_0 K$. Then, $\vec{v}_c$ and $\Delta M + E + q_0 K$ satisfy the following relation over the encoding mapping σ^{-1}:

$\sigma^{-1}(\vec{v}_c) = M_c = \sum_{i=0}^{n-1} (\Delta m_i + e_i + q_0 k_i) \cdot X^i$ # i.e., polynomial $\Delta M + E + q_0 K$

This implies that if we *homomorphically* apply the σ^{-1} transformation to the elements of the input vector $\vec{v}_c$, then the resulting input vector $\vec{v}_s$ will store $\vec{v}_c$'s encoded polynomial coefficient values as follows:

$\sigma^{-1} \circ \vec{v}_c = \vec{v}_s = (\Delta m_0 + e_0 + q_0 k_0,\ \Delta m_1 + e_1 + q_0 k_1,\ \cdots,\ \Delta m_{n-1} + e_{n-1} + q_0 k_{n-1})$

where $\circ$ represents a linear transformation operation comprising $(+, \cdot)$

However, remember that at the end of the ModRaise step, we get the decrypted (but corrupted by $q_0 k$) polynomial $M_c = \mathsf{RLWE}^{-1}(\mathsf{ct} = (A, B)) = \Delta M + E + q_0 K$ and we are not allowed to decode it into $\vec{v}_c$. Therefore, we will instead *encode* the matrix $\tilde{W} \cdot I_n^R$ in the encoding transformation σ^{-1} $(\vec{m} = \frac{\tilde{W} \cdot I_n^R \cdot \vec{v}}{n})$ into its equivalent polynomials (treating a matrix as a combination of vectors) and then perform batched slot-wise $(+, \cdot)$ operation between M_c and the polynomial version of $\tilde{W} \cdot I_n^R$. We express this polynomial-based computation as follows:

$M_s = \sigma^{-1}_{\sigma^{-1}} \circ M_c \bmod q_L$ # $\sigma^{-1}_{\sigma^{-1}}$ is the polynomial-encoded version of the σ^{-1} transformation

Then, the resulting polynomial M_s's corresponding input vector slots (i.e., the decoded version of M_s) will store $\vec{v}_s = (\Delta m_0 + e_0 + q_0 k_0,\ \Delta m_1 + e_1 + q_0 k_1,\ \cdots,\ \Delta m_{n-1} + e_{n-1} + q_0 k_{n-1})$. In other words, the above computation effectively *moves* the coefficients of M_c to the input vector slots of a new plaintext polynomial.

However, remember that in CKKS, an input vector can store only up to $\frac{n}{2}$ slots, whereas we need to store a total of n coefficients of M_c in the input vector slots. Therefore, we technically need to create 2 pieces of M_s as M_{s1} and M_{s2}, where the input vector of M_{s1} stores $(\Delta m_0 + e_0 + q_0 k_0,\ \Delta m_1 + e_1 + q_0 k_1,\ \cdots,\ \Delta m_{\frac{n}{2}-1} + e_{\frac{n}{2}-1} + q_0 k_{\frac{n}{2}-1})$, and the input vector of M_{s2} stores $(\Delta m_{\frac{n}{2}} + e_{\frac{n}{2}} + q_0 k_{\frac{n}{2}},\ \cdots,\ \Delta m_{n-1} + e_{n-1} + q_0 k_{n-1})$.

EvalExp: Our next step is to update $\vec{v}_s$'s each element $m_i + e_i + q_0 k_i$ to $m_i + e_i$ by evaluating it with the sine function $f(x)$. Since the output of the CoeffToSlot step is polynomial M_s (technically

M_{s1} and M_{s2}), we need to apply the evaluation transformation in an encoded form. First, we approximate $f(x)$ as a linear combination comprising only $(+, \cdot)$ operations by using the Taylor series and Euler's formula (will be explained later). Then, we encode (i.e., σ) the approximated formula into a polynomial form, and we denote it as σ_f. Finally, we apply the σ_f transformation to M_s as follows:

$\sigma_f^{-1} \circ M_s \bmod q_L$ # Applying the sine function's linear transformation to $\vec{v}_s$'s each slot storing $\Delta m_i + e_i + q_0 k_i$

$= M_t = \sigma(\vec{v}_t) = \sigma((\Delta m_i + e_i)_{i=0}^{n-1}) \bmod q_L$

After the linear transformation by the sine function, notice that each $q_0 k_i$ term gets eliminated from $\vec{v}_s$'s slots (i.e. modulo reduction by q) and the resulting vector $\vec{v}_t$ stores only the $\Delta m_i + e_i$ terms.

SlotToCoeff: Now that we have a polynomial M_t whose corresponding input vector $\vec{v}_t$'s slots store garbage-removed coefficients of (i.e., $\Delta m_i + e_i$) our initial plaintext polynomial, our next step is to put these coefficients stored in $\vec{v}_t$ back to the polynomial. This is an exact reverse operation of CoeffToSlot as follows:

$\sigma_\sigma^{-1} \circ M^t = M_b$ # σ_σ^{-1} is a polynomial-encoded form of the batch-decoding formula $\vec{v}' = \tilde{W}^* \cdot \vec{m}$ (??)

The result is polynomial M_b whose coefficients are garbage-eliminated (i.e., $q_0 k_i$-free) versions of M_c. Finally, we re-encrypt M_b as $\mathsf{RLWE}_{S,\sigma}(M_b)$ as the final modulus-bootstrapped ciphertext.

Bootstrapping Without a Secret Key: So far, we have assumed that we have access to the secret key S. With decryption and re-encryption enabled, the above bootstrapping steps described are mathematically equivalent to computing the following:

1. **INPUT:** $\mathsf{ct} = (A, B) \bmod q_0$ # where $\mathsf{ct} = (A, B) = \mathsf{RLWE}_{S,\sigma}(\Delta M)$
2. **ModRaise:** $\mathsf{ct} = (A, B) \bmod q_L$
3. **Decryption:** $\mathsf{RLWE}^{-1}(\mathsf{ct} = (A, B))) \bmod q_L$
4. **CoeffToSlot:** $\sigma_{\sigma^{-1}}^{-1} \circ \mathsf{RLWE}^{-1}(\mathsf{ct} = (A, B))) \bmod q_L$
5. **EvalExp:** $\sigma_f^{-1} \circ (\sigma_{\sigma^{-1}}^{-1} \circ \mathsf{RLWE}^{-1}(\mathsf{ct} = (A, B)))) \bmod q_L$
6. **SlotToCoeff:** $\sigma_\sigma^{-1} \circ (\sigma_f^{-1} \circ (\sigma_{\sigma^{-1}}^{-1} \circ \mathsf{RLWE}^{-1}(\mathsf{ct} = (A, B))))) \bmod q_L$
7. **Re-encryption:** $\mathsf{RLWE}_{S,\sigma}(\sigma_\sigma^{-1} \circ (\sigma_f^{-1} \circ (\sigma_{\sigma^{-1}}^{-1} \circ \mathsf{RLWE}^{-1}(\mathsf{ct} = (A, B)))))) \bmod q_L$

However, the ultimate goal of CKKS bootstrapping is to reset the modulus of a ciphertext from q_0 to q_L without having access to S.

Meanwhile, one important insight is that CKKS's ModRaise procedure on the ciphertext $(A, B) \bmod q_0$ from $q_0 \rightarrow q_L$ effectively transforms the ciphertext into a new one which is an encryption of $\Delta M + q_0 K$. Before ModRaise, ciphertext $(A, B) \bmod q_0$'s decryption relation is as follows:

$A \cdot A + B = \Delta M + E + K q_0 \bmod q_0 = \Delta M + E$

After ModRaise to $(A, B) \bmod q_L$, its decryption relation is as follows:

$A \cdot S + B = \Delta M + E + K q_0 \bmod q_L = \Delta M + E + K q_0$ # because $\Delta M + E + K q_0 \ll q_L$

Therefore, the *mod-raised* ciphertext $(A, B) \bmod q_L = \mathsf{RLWE}_{S,\sigma}(\Delta M + K q_0)$ with noise E. Thus,

CKKS's *homomorphic* bootstrapping strategy is to run the subsequent CoeffToSlot, EvalExp, and SlotToCoeff steps homomorphically based on the ciphertext (A, B) mod q_L. Running these 3 steps consumes a few multiplicative levels due to the ciphertext-to-ciphertext multiplication operations when homomorphically multiplying the coefficient-to-slot and slot-to-coefficient transformation matrices and homomorphically computing powers of X (i.e., X^k) during sine approximation. Therefore, upon completion of these 3 steps, the ciphertext modulus reduces from $q_L \rightarrow q_l$ (where l is some integer such that $l < L$).

Note that the result of homomorphic bootstrapping is equal to the explicit bootstrapping based on decryption & re-encryption (if we ignore the small differences in the final ciphertext modulus and the noise). In the following subsections, we will explain the algebraic details of CoeffToSlot, EvalExp and SlotToCoeff steps.

D-3.13.3 Details: CoeffToSlot

Homomorphically moving the coefficients of M_c (i.e., $\Delta m_i + e_i + q_0 k_i$ for $0 \leq i \leq n-1$) to a new ciphertext's input vector slots is mathematically equivalent to homomorphically computing $\sigma_{\sigma^{-1}}^{-1} \circ (\mathsf{RLWE}_{S,\sigma}(\mathsf{ct} = (A, B))$, which is equivalent to applying the encoding formula to the input vector slot values of $\mathsf{RLWE}_{S,\sigma}(\mathsf{ct} = (A, B))$.

As explained in Summary **??** (in **??**), the encoding formula for converting an input vector into a list of polynomial coefficients is $\vec{m} = \dfrac{\tilde{W} \cdot I_n^R \cdot \vec{v}}{n}$, where $\tilde{W}$ is a basis of the n-dimensional vector space crafted as follows:

$$\tilde{W} = \begin{bmatrix} 1 & 1 & \cdots & 1 & 1 & 1 & \cdots & 1 \\ (\omega^{J(\frac{n}{2}-1)}) & (\omega^{J(\frac{n}{2}-2)}) & \cdots & (\omega^{J(0)}) & (\omega^{J_*(\frac{n}{2}-1)}) & (\omega^{J_*(\frac{n}{2}-2)}) & \cdots & (\omega^{J_*(0)}) \\ (\omega^{J(\frac{n}{2}-1)})^2 & (\omega^{J(\frac{n}{2}-2)})^2 & \cdots & (\omega^{J(0)})^2 & (\omega^{J_*(\frac{n}{2}-1)})^2 & (\omega^{J_*(\frac{n}{2}-2)})^2 & \cdots & (\omega^{J_*(0)})^2 \\ \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \vdots \\ (\omega^{J(\frac{n}{2}-1)})^{n-1} & (\omega^{J(\frac{n}{2}-2)})^{n-1} & \cdots & (\omega^{J(0)})^{n-1} & (\omega^{J_*(\frac{n}{2}-1)})^{n-1} & (\omega^{J_*(\frac{n}{2}-2)})^{n-1} & \cdots & (\omega^{J_*(0)})^{n-1} \end{bmatrix}$$

\# where the rotation helper function $J(h) = 5^h \bmod 2n$

Therefore, given the input ciphertext $\mathsf{ct_c} = \mathsf{RLWE}_{S,\sigma}(\mathsf{ct} = (A, B))$ mod q_L whose plaintext polynomial M_c contains corrupted coefficients, computing $\sigma_{\sigma^{-1}}^{-1} \circ \mathsf{ct_c}$ is equivalent to computing $\dfrac{\tilde{W} \cdot I_n^R \cdot \mathsf{ct_c}}{n}$. However, one problem here is that each CKKS ciphertext encodes only $\dfrac{n}{2}$ input vector slots, whereas our goal is to move n (corrupted) coefficients of the plaintext polynomial M_c encrypted in $\mathsf{ct_c}$. Therefore, we will instead generate 2 ciphertexts, $\mathsf{ct_{s1}}$ and $\mathsf{ct_{s2}}$, such that each $\mathsf{ct_{s1}}$'s input vector slots store $(\Delta m_i + e_i + q_0 k_i)_{0 \leq i < \frac{n}{2}}$ and $\mathsf{ct_{s2}}$'s input vector slots store $(\Delta m_i + e_i + q_0 k_i)_{\frac{n}{2} \leq i < n}$.

We split the $n \times n$ matrix $\tilde{W} \cdot I_n^R$ into four $\dfrac{n}{2} \times \dfrac{n}{2}$ matrices as follows:

- $[\tilde{W} I_n^R]_{11}$: a matrix comprising the upper left-half section of $\tilde{W} \cdot I_n^R$
- $[\tilde{W} I_n^R]_{12}$: a matrix comprising the upper right-half section of $\tilde{W} \cdot I_n^R$
- $[\tilde{W} I_n^R]_{21}$: a matrix comprising the lower left-half section of $\tilde{W} \cdot I_n^R$
- $[\tilde{W} I_n^R]_{22}$: a matrix comprising the lower right-half section of $\tilde{W} \cdot I_n^R$

Then, we can compute $\mathsf{ct_{s1}}$ and $\mathsf{ct_{s2}}$ as follows:

$$\mathsf{ct_{s1}} = \frac{[\tilde{W} I_n^R]_{11} \cdot \mathsf{ct_c} + [\tilde{W} I_n^R]_{12} \cdot I_{\frac{n}{2}}^R \cdot \overline{\mathsf{ct_c}}}{n}$$

$$\mathsf{ct_{s2}} = \frac{[\tilde{W} I_n^R]_{21} \cdot \mathsf{ct_c} + [\tilde{W} I_n^R]_{22} \cdot I^R_{\frac{n}{2}} \cdot \overline{\mathsf{ct_c}}}{n}$$

Each homomorphic matrix-vector multiplication (e.g., $[\tilde{W} I_n^R]_{21} \cdot \mathsf{ct_c}$) can be done in an efficient manner that reduces the number of homomorphic rotations (**??**). $\overline{\mathsf{ct_c}}$ can be computed by applying homomorphic conjugation to $\mathsf{ct_c}$ (**??**).

D-3.13.4 Details: EvalExp

We will use the sine function $f(x) = \frac{q}{2\pi} \cdot \sin\left(\frac{2\pi x}{q_0}\right)$ to approximately eliminate $q_0 k_i$ from $\Delta m_i + e_i + q_0 k_i$ by computing $f(\Delta m_i + e_i + q_0 k_i) \approx \Delta m_i + e_i$. This approximation works if $\Delta m_i + e_i$ is very close to $x = 0$ relative to q_0 (i.e., $\Delta m_i + e_i \ll q_0$). Still, the elimination of $q_0 k_i$ is approximate (i.e., $\approx \Delta m_i + e_i$), because $f(x)$ is $y \approx x$ nearby $x = 0$, not exactly $y = x$.

One issue is that we need to evaluate $f(x)$ homomorphically based on $\mathsf{ct_{s1}}$ and $\mathsf{ct_{s2}}$ as inputs (i.e., $f(\mathsf{ct_{s1}})$ and $f(\mathsf{ct_{s2}})$), but FHE supports only $(+, \cdot)$ operations, whereas the sine graph cannot be formulated by only $(+, \cdot)$. Therefore, we will approximate the sine function $f(x)$ by using the Taylor series (**??**):

$$f(x) = f(a) + \frac{f'(a)}{1!}(x-a) + \frac{f''(a)}{2!}(x-a)^2 + \frac{f'''(a)}{3!}(x-a)^3 + \cdots = \sum_{d=0}^{\infty} \frac{f^{(d)}(a)}{d!}(x-a)^d$$

If we approximate $f(x)$ around $x = 0$, then the approximated polynomial is as follows:

$$\begin{aligned} f(x) &= \frac{q_0}{2\pi} \cdot \sin\left(\frac{2\pi}{q_0} \cdot 0\right) + \frac{q_0}{2\pi} \cdot \frac{2\pi}{q_0} \cdot \frac{\cos\left(\frac{2\pi}{q_0} \cdot 0\right)}{1!} \cdot x + \frac{q_0}{2\pi} \cdot \left(\frac{2\pi}{q_0}\right)^2 \cdot \frac{-\sin\left(\frac{2\pi}{q_0} \cdot 0\right)}{2!} \cdot x^2 + \cdots \\ &= \frac{q_0}{2\pi} \cdot \sum_{j=0}^{\infty} \left(\frac{(-1)^j}{(2j+1)!} \cdot \left(\frac{2\pi x}{q_0}\right)^{2j+1} \right) \\ &\approx \frac{q_0}{2\pi} \cdot \sum_{j=0}^{h} \left(\frac{(-1)^j}{(2j+1)!} \cdot \left(\frac{2\pi x}{q_0}\right)^{2j+1} \right) = \hat{f}(x) \end{aligned}$$

, where $\hat{f}(x)$ is a $(2h+1)$-degree polynomial.

Remember that in the RLWE cryptosystem, $B + AS \bmod q_0 = \Delta M + E$, or $B + AS = \Delta M + E + q_0 K$ with some polynomial K representing the wrapping around values of modulo q_0. Since the secret key S is an $(n-1)$-degree polynomial whose coefficients are small (i.e., $s_i \in \{-1, 0, 1\}$), the coefficients of K will have some *reasonably small* upper bound, which decreases with the sparsity of S (i.e., the frequency of 0 coefficients in S). Therefore, the degree of our approximated $\hat{f}(x)$ only needs to be high enough to accurately evaluate y values between $-q_0 \cdot k_{max} \le x \le q_0 \cdot k_{max}$. The required minimum degree of our approximated Taylor polynomial $\hat{f}(x)$ increases with $q_0 k_{max}$ (i.e., the upper bound of x). Our one issue is that the computation overhead for homomorphic evaluation of a polynomial generally increases exponentially with the degree of the polynomial, which will slow down bootstrapping. To reduce this computation cost, we will leverage Euler's formula (**??**) and its square arithmetic:

$$\begin{cases} e^{i\cdot\theta} = \cos\theta + i \cdot \sin\theta \\ (e^{i\cdot\theta})^2 = e^{i\cdot 2\theta} \end{cases}$$

By substituting $\theta = \frac{2\pi x}{q_0}$, we will use Euler's formula. We will also approximate $e^{i\theta}$ with the

Taylor series, but instead of directly approximating $e^{i\theta}$, we will first approximate $e^{\frac{i\theta}{2^r}}$ for some large 2^r. After that, we will iteratively square $e^{\frac{i\theta}{2^r}}$ a total r times. Then, we get an approximation of $(e^{\frac{i\theta}{2^r}})^{2^r} = e^{i\theta}$. The reason why we start with the approximation of $e^{\frac{i\theta}{2^r}}$ instead of $e^{i\theta}$ is that its approximation requires a small degree of polynomial, as $\frac{\theta}{2^r}$ (i.e., the input to the complex exponential function) is small provided 2^r is sufficiently large. Specifically, we learned that x ($= \Delta m_i + e_i + q_0 k_i$) is upper-bounded by $q_0 k_{max}$, thus $\theta = \frac{2\pi x}{q_0}$ is upper-bounded by $\frac{2\pi k_{max}}{2^r}$. As the targeted range of x for approximation in $f(x)$ is small, we need a small degree of Taylor series polynomial.

Using the Taylor series with degree d_0 around $x = 0$, we can approximate $e^{\frac{2\pi i x}{2^r q_0}}$ as:
$f_e(x) = e^{\frac{2\pi i x}{2^r q_0}} \approx \sum_{d=0}^{d_0} \frac{1}{d!} \left(\frac{2\pi i x}{2^r q_0}\right)^d = \hat{f}_e(x)$

Then, we iteratively square $\hat{f}_e$ total r times to get:
$(\hat{f}_e(x))^{2^r} \approx (f_e(x))^{2^r} = e^{i\frac{2\pi x}{q_0}} = e^{i\theta}$

Then, based on Euler's formula $e^{i\cdot\theta} = \cos\theta + i \cdot \sin\theta$, we can derive the following relations:

$\overline{e^{i\cdot\theta}} = \cos\theta + \overline{i \cdot \sin\theta}$
$e^{-i\cdot\theta} = \cos\theta - i \cdot \sin\theta$
$e^{i\cdot\theta} - e^{-i\cdot\theta} = (\cos\theta + i \cdot \sin\theta) - (\cos\theta - i \cdot \sin\theta) = 2i \sin\theta$
$\sin\theta = \frac{-i}{2} \cdot (e^{i\cdot\theta} - e^{-i\cdot\theta})$
$\frac{q_0}{2\pi} \cdot \sin\theta = \frac{q_0}{2\pi} \cdot \frac{-i}{2} \cdot (e^{i\cdot\theta} - e^{-i\cdot\theta})$

Substituting $\theta = \frac{2\pi x}{q_0}$, we finally get:
$\frac{q_0}{2\pi} \cdot \sin\left(\frac{2\pi x}{q_0}\right) = \frac{q_0}{2\pi} \cdot \frac{-i}{2} \cdot (e^{i\cdot\frac{2\pi x}{q_0}} - e^{-i\cdot\frac{2\pi x}{q_0}})$

Using the final relation above, the EvalExp step homomorphically evaluates the approximation of $\frac{q_0}{2\pi} \cdot \sin\left(\frac{2\pi x}{q_0}\right)$ where $x = \Delta m_i + e_i + q_0 k_i$ as follows:

1. Homomorphically approximately compute $\hat{f}(x) = e^{i\cdot\frac{2\pi x}{q_0}}$.
2. Homomorphically approximately compute $\overline{\hat{f}(x)} = e^{-i\cdot\frac{2\pi x}{q_0}}$ by applying homomorphic conjugation. (**??**) to $\hat{f}(x)$
3. Homomorphically compute $\hat{f}(x) - \overline{\hat{f}(x)} = e^{i\cdot\frac{2\pi x}{q_0}} - e^{-i\cdot\frac{2\pi x}{q_0}}$, and then multiply the result by $\frac{-i}{2}$ encoded as CKKS plaintext.

The result of EvalExp is two ciphertexts whose input vector slots store the bootstrapped coefficients of M_c, which are modulo-reduced q_0 from $\Delta m_i + e_i + q_0 k_i$ to $\Delta m_i + e_i + e_{bi}$. Note that e_{bi} is a bootstrapping error introduced by the following three factors: (1) the intrinsic homomorphic $(+, \cdot)$ computation noises of the CoeffToSlot, EvalExp, and SlotToCoeff steps; (2) the EvalExp step's Taylor polynomial approximation error of the exponential function $e^{i\theta}$; (3) the EvalExp step's sine graph error, since the graph is not exactly $y = x$ around $x = 0$, but only $y \approx x$.

Note that since the output of the $\mathsf{CoeffToSlot}$ step was split into 2 ciphertexts ($\mathsf{ct_{s1}}$ and $\mathsf{ct_{s2}}$), the output of the $\mathsf{EvalExp}$ step is also in 2 ciphertexts: ($\mathsf{ct_{b1}}$ and $\mathsf{ct_{b2}}$). The input vector slots of $\mathsf{ct_{b1}}$ store $(\Delta m_i + e_i + e_{bi})_{i=0}^{\frac{n}{2}-1}$, whereas the input vector slots of $\mathsf{ct_{b2}}$ store $(\Delta m_i + e_i + e_{bi})_{i=\frac{n}{2}}^{n-1}$.

D-3.13.5 Details: SlotToCoeff

This step is an exact inverse of the $\mathsf{CoeffToSlot}$ step, which is moving the bootstrapped (i.e. modulo-reduced q_0) coefficients of M_v stored in the input vector slots back to the final plaintext polynomial M_f. Remember that the decoding formula from a polynomial to an input vector (**??**) is $\vec{v'} = \tilde{W}^* \cdot \vec{m}$, where:

$$\tilde{W}^* = \begin{bmatrix} 1 & (\omega^{J(0)}) & (\omega^{J(0)})^2 & \cdots & (\omega^{J(0)})^{n-1} \\ 1 & (\omega^{J(1)}) & (\omega^{J(1)})^2 & \cdots & (\omega^{J(1)})^{n-1} \\ 1 & (\omega^{J(2)}) & (\omega^{J(2)})^2 & \cdots & (\omega^{J(2)})^{n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & (\omega^{J(\frac{n}{2}-1)}) & (\omega^{J(\frac{n}{2}-1)})^2 & \cdots & (\omega^{J(\frac{n}{2}-1)})^{n-1} \\ 1 & (\omega^{J_*(\frac{n}{2}-1)}) & (\omega^{J_*(\frac{n}{2}-1)})^2 & \cdots & (\omega^{J_*(\frac{n}{2}-1)})^{n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & (\omega^{J_*(1)}) & (\omega^{J_*(1)})^2 & \cdots & (\omega^{J_*(1)})^{n-1} \\ 1 & (\omega^{J_*(0)}) & (\omega^{J_*(0)})^2 & \cdots & (\omega^{J_*(0)})^{n-1} \end{bmatrix}$$

We denote $[\tilde{W}^*]_{11}$, $[\tilde{W}^*]_{12}$, $[\tilde{W}^*]_{21}$, and $[\tilde{W}^*]_{22}$ as $\frac{n}{2} \times \frac{n}{2}$ matrices corresponding to the upper-left, upper-right, lower-left, and lower-right sections of $\tilde{W}^*$. Then, homomorphically applying the decoding formula results in the final bootstrapped ciphertext $\mathsf{ct_{final}}$ modulo q_L whose plaintext polynomial is garbage-eliminated from $\Delta M + E + q_0 K \bmod q_L$ to $\Delta M + E + E_b \bmod q_l$ (where E_b is the bootstrapping error polynomial). Note that the ciphertext modulus changed from $q_L \rightarrow q_l$ because we consumed some multiplicative levels for computing ciphertext-to-ciphertext multiplications during polynomial evaluation (i.e., X^k). We can derive $\mathsf{ct_{final}}$ by homomorphically applying the decoding transformation $\tilde{W}_T$ to the results of the $\mathsf{EvalExp}$ step ($\mathsf{ct_{b1}}$ and $\mathsf{ct_{b2}}$) as follows:

$\mathsf{ct_{final}} = \mathsf{RLWE}_{S,\sigma}(\Delta M + E + E_b) = [\tilde{W}^*]_{11} \cdot \mathsf{ct_{b1}} + [\tilde{W}^*]_{12} \cdot \mathsf{ct_{b2}}$

Note that we do not use $[\tilde{W}^*]_{21}$ and $[\tilde{W}^*]_{22}$, because we only need to derive the $\frac{n}{2}$ input vector slots whose decoding would result in the n coefficients $(\Delta m_i + e_i + e_{bi})_{i=0}^{n-1}$ of the final $(n-1)$-degree polynomial. Once we generate a new ciphertext $\mathsf{ct_{final}}$ whose $\frac{n}{2}$ input vector slots store $(\Delta m_i + e_i + e_{bi})_{i=0}^{n-1}$, then its latter $\frac{n}{2}$ conjugate slots get automatically filled with the conjugates of the first $\frac{n}{2}$ slot values.

D-3.13.6 Reducing the Bootstrapping Overhead by Sparsely Packing Ciphertext

In many cases, the application of CKKS may use only a small number of input vector slots (e.g., $\frac{n'}{2}$) out of $\frac{n}{2}$ slots. Suppose that such n' is some number that divides n. Then, we can do a series of homomorphic rotations and multiplications to make the input vector slots store $\frac{n}{n'}$ repetitions

of the $\frac{n'}{2}$-slot values. Specifically, we can do this in total $\frac{n}{n'}$ rounds of rotations and additions: initially, we zero-mask between the $\frac{n'}{2}$-th slot and the $\frac{n}{2}-1$-th slots and save as ct, and then in each i-th round we compute $\mathsf{ct} = \mathsf{ct} + \mathsf{Rotate}(\mathsf{ct}, -n' \cdot 2^i)$.

Then, we apply the optimization of sparsely packing ciphertext in Summary **??** (**??**): if an $\frac{n}{2}$-dimensional input vector is structured as $\frac{n}{n'}$ consecutive repetitions of the first $\frac{n'}{2}$ slot values, then its encoded polynomial $M(X) \in \mathbb{Z}[X]/(X^n+1)$ has the structure such that all its coefficients whose degree term is not a multiple of $\frac{n}{n'}$ are zero as follows:

$M(X) = c_0 + c_{\frac{n}{n'}} X^{\frac{n}{n'}} + c_{\frac{2n}{n'}} X^{\frac{2n}{n'}} + \cdots + c_{n-\frac{n}{n'}} X^{n-\frac{n}{n'}}$.

Remember that in the $\mathsf{CoeffToSlot}$ step (**??**), we use the formula $\vec{m} = \frac{\tilde{W} \cdot I_n^R \cdot \vec{v'}}{n}$ to move the $q_0 k$-contaminated polynomial's coefficients to the input vector slots. But by the principle of sparsely packed ciphertext, we know that all the slots of $\vec{m}$ which are not a multiple of $\frac{n}{n'}$ slots would store a zero coefficient. This means that we will get the same computation result even if we only compute the $\vec{m} = \frac{\tilde{W} \cdot I_n^R \cdot \vec{v'}}{n}$ formula with the rows of $\tilde{W}$ whose row index is a multiple of $\frac{n}{n'}$. Mathematically, we can update the encoding formula to $\vec{m_s} = \frac{\Im \cdot I_{n'}^R \cdot \vec{v'}}{n'}$ where the $n \times \frac{n}{n'}$ matrix $\Im$ is an elimination of all those columns from $\tilde{W}$ whose column index is not a multiple of $\frac{n}{n'}$:

$$\Im = \begin{bmatrix} 1 & 1 & \cdots & 1 & 1 & \cdots & 1 & 1 \\ (\xi^{J(\frac{0\cdot n}{n'}-n')}) & (\xi^{J(\frac{1\cdot n}{n'})}) & \cdots & (\xi^{J(n-\frac{n}{n'})}) & (\xi^{J_*(n-\frac{n}{n'})}) & \cdots & (\xi^{J_*(\frac{1\cdot n}{n'})}) & (\xi^{J_*(\frac{0\cdot n}{n'}-n')}) \\ (\xi^{J(\frac{0\cdot n}{n'}-n')})^2 & (\xi^{J(\frac{1\cdot n}{n'}-n')})^2 & \cdots & (\xi^{J(n-\frac{n}{n'})})^2 & (\xi^{J_*(n-\frac{n}{n'})})^2 & \cdots & (\xi^{J_*(\frac{1\cdot n}{n'}-n')})^2 & (\xi^{J_*(\frac{0\cdot n}{n'}-n')})^2 \\ \vdots & \vdots & \ddots & \vdots & \vdots & \ddots & \vdots & \vdots \\ (\xi^{J(\frac{0\cdot n}{n'}-n')})^{n-1} & (\xi^{J(\frac{1\cdot n}{n'}-n')})^{n-1} & \cdots & (\xi^{J(n-\frac{n}{n'})})^{n-1} & (\xi^{J_*(n-\frac{n}{n'})})^{n-1} & \vdots & (\xi^{J_*(\frac{1\cdot n}{n'}-n')})^{n-1} & (\xi^{J_*(\frac{0\cdot n}{n'}-n')})^{n-1} \end{bmatrix}$$

Remember that in the original $\mathsf{CoeffToSlot}$ step (**??**), we had to split $\mathsf{ct_s}$ into $\mathsf{ct_{s1}}$ and $\mathsf{ct_{s2}}$ because in CKKS each input vector can store a maximum of $\frac{n}{2}$ slots but we need to move a total of n coefficient values to the input vector slots for bootstrapping. On the other hand, the computation result of the above updated encoding formula (using a sparsely packed ciphertext) is $\vec{m}_s$, having only $\frac{n}{n'}$ coefficient slots instead of n coefficient slots, and each slot index i in $\vec{m}_s$ corresponds to the encoded polynomial's coefficient with degree term $i \cdot \frac{n}{n'}$ (we do not compute any other coefficient terms, because we know that they are 0 anyway, so no need to bootstrap them). And notice that $\frac{n}{n'} \leq \frac{n}{2}$, because n' divides n. Therefore, without computing two ciphertexts $\mathsf{ct_{s1}} = [\tilde{W} I_n^R]_{11} \cdot \mathsf{ct_c} + [\tilde{W} I_n^R]_{12} \cdot I_{\frac{n}{2}}^R \cdot \overline{\mathsf{ct_c}}$ and $\mathsf{ct_{s2}} = [\tilde{W} I_n^R]_{21} \cdot \mathsf{ct_c} + [\tilde{W} I_n^R]_{22} \cdot I_{\frac{n}{2}}^R \cdot \overline{\mathsf{ct_c}}$ separately, we can directly compute $\mathsf{ct_c} = \frac{\Im \cdot I_{n'}^R \cdot \mathsf{ct_c}}{n'}$, because all coefficients for bootstrapping fit in $\frac{n}{2}$ slots. Therefore, the number of homomorphic computations and memory requirement for the $\mathsf{CoeffToSlot}$ step can be reduced by half. And the same is true for the $\mathsf{EvalExp}$ step (**??**).

Similarly, as for the $\mathsf{SlotToCoeff}$ step (**??**), we update the decoding formula $\vec{v'} = \tilde{W}^* \cdot \vec{m}$ to $\vec{v'} = \Im^T \cdot \vec{m}_c$. This again reduces the number of homomorphic computations and memory requirements for the $\mathsf{SlotToCoeff}$ step by half. Notice that $\Im^T$ is a matrix where those columns whose column index is not a multiple of $\frac{n}{n'}$ are zero. This zero-enforcement to the columns of $\Im^T$ still outputs the same computation result, because $\vec{m}_c$ is a vector such that those slots whose slot index is not a

multiple of $\frac{n}{n'}$ are zero, which makes the computation result with their corresponding columns of $\exists^T$ (i.e., the columns whose index is not a multiple of $\frac{n}{n'}$) zero, anyway.

D-3.13.7 Summary

We summarize the CKKS bootstrapping procedure as follows.

⟨Summary ??⟩ CKKS Bootstrapping

1. **INPUT:** $\mathsf{ct} = (A, B) \bmod q_0$ # where $\mathsf{ct} = (A, B) = \mathsf{RLWE}_{S,\sigma}(\Delta M)$

   , which satisfies the decryption relation: $A \cdot S + B = \Delta M + E + Kq_0$

2. **ModRaise:** View the polynomials A and B as plaintext polynomials whose each coefficient is in $\mathbb{Z}_{q_L}$ (i.e., $(A, B) \bmod q_L$). This change of viewpoint automatically changes the ciphertext as $\mathsf{RLWE}_{S,\sigma}(\Delta M + Kq_0)$. The ModRaise step does not require any actual computation.

3. **CoeffToSlot:**
   Move the coefficients of the encrypted plaintext $\Delta M + E + q_0 K$ to the input vector slots by homomorphically multiplying $n^{-1} \cdot \tilde{W} \cdot I_n^R$ to it follows:
   $\mathsf{RLWE}_{S,\sigma}(Z_1) = n^{-1} \cdot \tilde{W} \cdot I_n^R \cdot \mathsf{RLWE}_{S,\sigma}(\Delta M + E + q_0 K) \bmod q_L$

4. **EvalExp:**
   Remove the wrap-around garbage value $q_0 K$ in $\Delta M + E + q_0 K$ by homomorphically evaluating the polynomial σ_f which approximates a sine function with period q_0 as follows:
   $\mathsf{RLWE}_{S,\sigma}(Z_2) = \sigma_f \circ \mathsf{RLWE}_{S,\sigma}(Z_1) \bmod q_l$

   This step is equivalent to *homomorphically* performing modulo reduction by q_0 to the input value. This step reduces the ciphertext modulus from $q_L \to q_l$ as it consumes multiplicative levels when homomorphically evaluating the polynomial approximation of the sine function.

5. **SlotToCoeff:**
   Move the modulo-q_0-reduced plaintext value $\Delta M + E$ stored in the input vector slots back to the plaintext coefficient positions by homomorphically multiplying the encoding matrix $\tilde{W}^*$ as follows:
   $\mathsf{RLWE}_{S,\sigma}(\Delta M + E) = \tilde{W}^* \cdot \mathsf{RLWE}_{S,\sigma}(Z_2) \bmod q_l$

Limitation: The noise slowly grows over each bootstrapping due to the bootstrapping error and will eventually overflow the message and the ciphertext modulus.

Comparison between BFV and CKKS Bootstrapping: In the case of CKKS's bootstrapping, it does not reduce the magnitude of the old noise E and keeps it the same as before, because the sine

approximation function converts $\Delta M + E + Kq_0$ into $\Delta M + E$. However, as the ciphertext modulus gets increased from $q_0 \to q_L$, the noise-to-ciphertext-modulus ratio decreases, since $\frac{E}{q_L} \ll \frac{E}{q_0}$. On the other hand, the bootstrapping procedure introduces a new bootstrapping noise, which can be viewed as a fixed amount. However, this fixed amount of new noise accumulates over each bootstrapping. Therefore, after a very large number of bootstrappings, the noise will eventually overflow the message and even the ciphertext modulus.

In the case of BFV's bootstrapping, it reduces the noise, but does not change the ciphertext modulus. However, there is no need to reset the ciphertext modulus, because BFV does not have a leveled ciphertext modulus chain, and BFV's ciphertext-to-ciphertext multiplication does not consume ciphertext modulus. Furthermore, since BFV's bootstrapping directly removes the noise, the noise is guaranteed to be kept under a certain threshold even after an infinite number of bootstrappings.

Another important difference is that CKKS's bootstrapping does not require homomorphic decryption, primarily because it maintains the plaintext's scaling factor to be the same across the entire bootstrapping procedure. On the other hand, BFV's bootstrapping needs to change the plaintext's scaling factor to run the digit extraction algorithm. Therefore, homomorphic decryption is required to change the plaintext scaling factor (p^e) while preserving the same ciphertext modulus (q).

D-3.13.8 Reducing the Bootstrapping Noise

As explained in **??**, the bootstrapping procedure generates three types of noises:

- Type-1 Noise: the intrinsic homomorphic $(+, \cdot)$ computation noises of the CoeffToSlot, EvalExp, and SlotToCoeff steps
- Type-2 Noise: the EvalExp step's approximation error of the exponential function $e^{i\theta}$
- Type-3 Noise: the EvalExp step's sine graph error (i.e., not exactly $y = x$ around $x = 0$, but only $y \approx x$)

The Type-1 noise is inevitable by the design of FHE. The Type-2 noise can be either avoided or unavoidable depending on the tradeoff setup between the bootstrapping accuracy and efficiency. Unlike these two types of noises, the Type-3 noise can be effectively reduced by newer bootstrapping techniques.

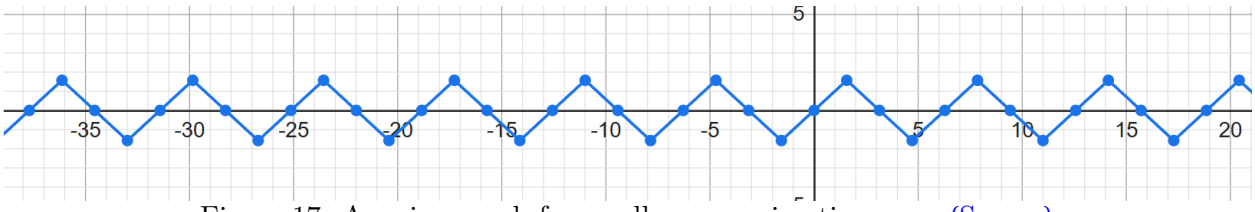


Figure 17: Arc-sine graph for smaller approximation error (Source)

arcsin(sin(x)) Approximation (EUROCRYPT 2021 [?]): Using the arcsin(sin(x)) function instead of the $\sin x$ function can reduce the Type-3 noise, because its line is not curved but straight, as shown in **??** (comprising a series of $y = x$ and $y = -x$ segments). This technique also uses the Remez algorithm that evenly distributes the approximation error over a specified region. However,

one downside of this technique is that it consumes 3 multiplicative levels.

Meta-BTS (CCS 2022 [?]): This is thus far the most computationally efficient and accurate bootstrapping technique, whose procedure is as follows:

1. Perform the regular bootstrapping based on the sine graph to the input ciphertext.
2. Rescale step 1's bootstrapped ciphertext to modulus q_0.
3. Subtract step 2's ciphertext from the initial un-bootstrapped ciphertext (where both ciphertexts are modulo q_0), whose result is a modulo q_0 ciphertext storing the bootstrapping error.
4. Bootstrap the output ciphertext of step 3 (storing the bootstrapping error) to modulus q_l.
5. Subtract step 4's ciphertext from step 1's ciphertext (where both ciphertexts are modulo q_l), which gives a new modulo q_l ciphertext with a reduced bootstrapping error.

Limitation in Noise Handling: The above bootstrapping techniques can reduce the Type-3 noise, because the bootstrapping error is smaller than the plaintext message and a smaller input x value to the approximating sine function outputs a value closer to $y = x$. Running this algorithm multiple times, the Type-3 noise becomes exponentially smaller, because the size of the target plaintext (i.e., the extracted bootstrapping error as the output of step 3 above) is much smaller than before. Meanwhile, Type-1 and Type-2 noises do not decrease over multiple bootstrapping rounds, relatively keeping their same level, because each round generates new Type-1 and Type-2 noises.

D-4 BGV Scheme

Similar to BFV, the BGV scheme is designed for homomorphic addition and multiplication of integers. Unlike CKKS, BGV guarantees exact encryption and decryption. From this view, BGV is similar to BFV. However, the major difference between these two schemes is that BFV stores the plaintext value in the MSBs (most significant bits) and the noise in the low-digit area (least significant bits), while BGV stores them the other way around: the plaintext value in the low-digit area and the noise in the MSBs. Technically, while BFV scales the plaintext polynomial by Δ, BGV scales the noise polynomial by Δ. Therefore, these two schemes use slightly different strategies to store and manage the plaintext and noise within a ciphertext.

BGV internally uses almost the same strategy as BFV for plaintext encoding, ciphertext-to-plaintext addition, ciphertext-to-ciphertext addition, ciphertext-to-plaintext multiplication, and input vector rotation. On the other hand, BGV's encryption and decryption are slightly different from BFV's scheme, because its scaling target is not the plaintext, but the noise. Also, unlike BFV where ciphertext-to-ciphertext multiplication has no limit on the number, BGV's ciphertext-to-ciphertext multiplication is leveled, switching the modulus to a lower level like CKKS, and thus it is limited. Furthermore, BGV's modulus switch and bootstrapping are partially different from BFV's.

Required Background

- ??: ??
- ??: ??
- ??: ??
- ??: ??
- ??: ??
- ??: ??
- ??: ??
- ??: ??
- ??: ??
- ??: ??
- ??: ??
- ??: ??
- ??: ??
- ??: ??
- ??: ??
- ??: ??
- ??: ??
- ??: ??
- ??: ??
- ??: ??
- ??: ??
- ??: ??
- ??: ??
- ??: ??
- ??: ??

D-4.1 Encoding and Decoding

BGV uses almost the same plaintext encoding scheme as BFV as described in Summary **??** in **??**, with the only difference that the scaling factor $\Delta = \frac{q_0}{t}$ is not applied to the plaintext polynomial $M(X)$ like BFV does. Instead, BGV applies its own scaling factor $\Delta = t$ to the noise polynomial $E(X)$ whenever it encrypts a new ciphertext (will be explained in **??**).

The following is BGV's encoding and decoding scheme.

⟨Summary ??⟩ BGV's Encoding and Decoding

Input: An n-dimensional integer modulo t vector $\vec{v} = (v_0, v_1, \cdots, v_{n-1}) \in \mathbb{Z}_t^n$

Encoding:

Convert $\vec{v} \in \mathbb{Z}_t^n$ into $\vec{m} \in \mathbb{Z}_t^n$ by applying the transformation $\vec{m} = \dfrac{\tilde{W} \cdot I_n^R \cdot \vec{v}}{n}$

, where $\tilde{W}$ is a basis of the n-dimensional vector space crafted as follows:

$$\tilde{W} = \begin{bmatrix} 1 & 1 & \cdots & 1 & 1 & 1 & \cdots & 1 \\ (\omega^{J(\frac{n}{2}-1)}) & (\omega^{J(\frac{n}{2}-2)}) & \cdots & (\omega^{J(0)}) & (\omega^{J_*(\frac{n}{2}-1)}) & (\omega^{J_*(\frac{n}{2}-2)}) & \cdots & (\omega^{J_*(0)}) \\ (\omega^{J(\frac{n}{2}-1)})^2 & (\omega^{J(\frac{n}{2}-2)})^2 & \cdots & (\omega^{J(0)})^2 & (\omega^{J_*(\frac{n}{2}-1)})^2 & (\omega^{J_*(\frac{n}{2}-2)})^2 & \cdots & (\omega^{J_*(0)})^2 \\ \vdots & \vdots & \ddots & \vdots & \vdots & \ddots & \vdots & \vdots \\ (\omega^{J(\frac{n}{2}-1)})^{n-1} & (\omega^{J(\frac{n}{2}-2)})^{n-1} & \cdots & (\omega^{J(0)})^{n-1} & (\omega^{J_*(\frac{n}{2}-1)})^{n-1} & (\omega^{J_*(\frac{n}{2}-2)})^{n-1} & \vdots & (\omega^{J_*(0)})^{n-1} \end{bmatrix}$$

where $\omega = g^{\frac{t-1}{2n}} \bmod t$ (g is a generator of $\mathbb{Z}_t^\times$)

The final output is $M = \sum_{i=0}^{n-1} m_i X^i \in \mathbb{Z}_t[X]/(X^n+1)$, which we can also treat as $M = \sum_{i=0}^{n-1} m_i X^i \in \mathbb{Z}_q[X]/(X^n+1)$ during encryption/decryption later, because the initial fresh coefficients m_i are guaranteed to be smaller than any q where $q = \{q_0, q_1, \cdots, q_L\}$.

Decoding: For the plaintext polynomial $M = \sum_{i=0}^{n-1} m_i X^i$, compute $\vec{v} = \tilde{W}^* \cdot \vec{m}$, where

$$\tilde{W}^* = \begin{bmatrix} 1 & (\omega^{J(0)}) & (\omega^{J(0)})^2 & \cdots & (\omega^{J(0)})^{n-1} \\ 1 & (\omega^{J(1)}) & (\omega^{J(1)})^2 & \cdots & (\omega^{J(1)})^{n-1} \\ 1 & (\omega^{J(2)}) & (\omega^{J(2)})^2 & \cdots & (\omega^{J(2)})^{n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & (\omega^{J(\frac{n}{2}-1)}) & (\omega^{J(\frac{n}{2}-1)})^2 & \cdots & (\omega^{J(\frac{n}{2}-1)})^{n-1} \\ 1 & (\omega^{J_*(0)}) & (\omega^{J_*(0)})^2 & \cdots & (\omega^{J_*(0)})^{n-1} \\ 1 & (\omega^{J_*(1)}) & (\omega^{J_*(1)})^2 & \cdots & (\omega^{J_*(1)})^{n-1} \\ 1 & (\omega^{J_*(2)}) & (\omega^{J_*(2)})^2 & \cdots & (\omega^{J_*(2)})^{n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & (\omega^{J_*(\frac{n}{2}-1)}) & (\omega^{J_*(\frac{n}{2}-1)})^2 & \cdots & (\omega^{J_*(\frac{n}{2}-1)})^{n-1} \end{bmatrix}$$

D-4.2 Encryption and Decryption

BGV's encryption and decryption scheme is very similar to BFV's scheme (Summary **??** in **??**) with a small difference: while BFV scales the plaintext polynomial $M(X)$ by Δ, BGV scales the noise polynomial $E(X)$ by Δ. In BFV, each encoded plaintext polynomial $M(X)$ is scaled by $\Delta = \left\lfloor \frac{q}{t} \right\rfloor$. This strategy effectively shifts each plaintext coefficient value to the most significant bits while keeping the noise in the least significant bits. On the other hand, BGV does not scale the plaintext polynomial $M(X)$, but instead it scales each new noise $E(X)$ by $\Delta = t$, making the noise $\Delta E(X)$, which is newly generated upon each new ciphertext creation. This different scaling strategy effectively shifts the noise (i.e., e_i) to the most significant bits by scaling it by $\Delta = t$ while keeping the plaintext value (i.e., m_i) $M(X)$'s each coefficient in the least significant bits.

Also, in BGV, the ciphertext modulus q is leveled like CKKS's one: $q \in \{q_0, q_1, \cdots, q_L\}$, where each $q_l = \prod_{i=0}^{l} w_i$ (where each w_i is a CRT modulus).

BGV's encryption decryption process is described as follows:

⟨Summary ??⟩ BGV Encryption and Decryption

Initial Setup:

- The plaintext modulus $t = p$ (a prime)
- The ciphertext modulus q is leveled like in CKKS: $q \in \{q_0, q_1, \cdots, q_L\}$, where each $q_l = \prod_{i=0}^{l} w_i$ (each w_i is a CRT modulus), and each $q_l \equiv 1 \bmod t$ (will be explained in **??**)
- The noise scaling factor $\Delta = t$
- The secret key $S \xleftarrow{\$} \mathcal{R}_{\langle n,2 \rangle}$. They coefficients of polynomial S can be either binary (i.e., $\{0,1\}$) or ternary (i.e., $\{-1,0,1\}$).

Encryption Input: $M \in \mathcal{R}_{\langle n,q \rangle}$, $A \xleftarrow{\$} \mathcal{R}_{\langle n,q \rangle}$, $E \xleftarrow{\chi_\sigma} \mathcal{R}_{\langle n,q \rangle}$

1. Compute $B = -A \cdot S + M + \Delta E \ \in \mathcal{R}_{\langle n,q \rangle}$
2. $\mathsf{RLWE}_{S,\sigma}(M + \Delta E) = (A, B) \ \in \mathcal{R}^2_{\langle n,q \rangle}$

Decryption Input: $C = (A, B) \ \in \mathcal{R}^2_{\langle n,q \rangle}$

1. $\mathsf{RLWE}^{-1}_{S,\sigma}(C) = B + A \cdot S = M + \Delta E \pmod q$
2. $M = M + \Delta E \bmod t$ # modulo reduction of $M + \Delta E$ by t

The final output is $M(X) \in \mathbb{Z}_t[X]/(X^n + 1)$

Conditions for Correct Decryption:

Each coefficient $\Delta e_i + m_i$ that contains the scaled noise and the plaintext should not overflow or underflow its ciphertext's any current moment's multiplicative level l's ciphertext modulus q_l (i.e., $\Delta e_i + m_i < q_l$)

When restoring the plaintext at the end of the decryption process, while BFV shifts down the plaintext and the noise to the lower bit area (which effectively rounds off the noise), BGV computes $\bmod p$, which effectively modulo-reduces the accumulated noise because every coefficient of E is a multiple of t (i.e., Δ). Finally, only the plaintext polynomial's each coefficient m_i remains in the

low-digit area without any noise e_i.

D-4.3 Ciphertext-to-Ciphertext Addition

BGV's ciphertext-to-ciphertext addition scheme is exactly the same as BFV's scheme (Summary **??** in **??**).

⟨Summary ??⟩ BGV Ciphertext-to-Ciphertext Addition

$$\begin{aligned}
&\mathsf{RLWE}_{S,\sigma}(M^{\langle 1\rangle} + \Delta E^{\langle 1\rangle}) + \mathsf{RLWE}_{S,\sigma}(M^{\langle 2\rangle} + \Delta E^{\langle 1\rangle}) \\
&= (A^{\langle 1\rangle},\ B^{\langle 1\rangle}) + (A^{\langle 2\rangle},\ B^{\langle 2\rangle}) \\
&= (A^{\langle 1\rangle} + A^{\langle 2\rangle},\ B^{\langle 1\rangle} + B^{\langle 2\rangle}) \\
&= \mathsf{RLWE}_{S,\sigma}\big((M^{\langle 1\rangle} + M^{\langle 2\rangle}) + \Delta E^{\langle 1\rangle} + \Delta E^{\langle 2\rangle}\big)
\end{aligned}$$

D-4.4 Ciphertext-to-Plaintext Addition

BGV's ciphertext-to-plaintext addition scheme is almost the same as BFV's scheme (Summary **??** in **??**). However, one difference is that it's not the case that the plaintext polynomial $\Lambda(X)$ to be added is scaled up by Δ, but it remains as $\Lambda(X)$.

⟨Summary ??⟩ BGV Ciphertext-to-Plaintext Addition

$$\begin{aligned}
&\mathsf{RLWE}_{S,\sigma}(M + \Delta E) + \Lambda \\
&= (A,\ B) + \Lambda \\
&= (A,\ B + \Lambda) \\
&= \mathsf{RLWE}_{S,\sigma}\big((M + \Lambda) + \Delta E\big)
\end{aligned}$$

D-4.5 Ciphertext-to-Plaintext Multiplication

BGV's ciphertext-to-plaintext multiplication scheme is exactly the same as BFV's scheme (Summary **??** in **??**).

⟨Summary ??⟩ BGV Ciphertext-to-Plaintext Multiplication

$$\begin{aligned}
&\mathsf{RLWE}_{S,\sigma}(M + \Delta E) \cdot \Lambda \\
&= (A,\ B) \cdot \Lambda \\
&= (A \cdot \Lambda,\ B \cdot \Lambda) \\
&= \mathsf{RLWE}_{S,\sigma}((M \cdot \Lambda) + \Delta E \cdot \Lambda)
\end{aligned}$$

Notice that BGV's ciphertext-to-plaintext multiplication does not consume any multiplicative level.

D-4.6 ModDrop

BGV's ModDrop works similarly to that of CKKS's ModDrop (Summary **??** in **??**). Remember that CKKS's ciphertext decryption relation is as follows:

$M + \Delta E = A \cdot S + B \bmod q_l$
$M + \Delta E = A \cdot S + B - K \cdot q_l$ # where $K \cdot q_l$ represents a modulo reduction by q_l

BGV's ModDrop operation decreases its modulus from $q_l \rightarrow q_{l-1}$ is performed by updating the ciphertext (A, B) to a new one: $(A' = A \bmod q_{l-1},\ B' = B \bmod q_{l-1})$. After the ModDrop, the ciphertext's modulus decreases from $q_l \rightarrow q_{l-1}$, yet its decryption relation still holds the same as follows:

$A' \cdot S + B' - K \cdot q_l$
$= (A \bmod q_{l-1}) \cdot S + (B \bmod q_{l-1}) - K \cdot q_l$
$= (A - K_A \cdot q_{l-1}) \cdot S + (B - K_B \cdot q_{l-1}) - K \cdot q_l$
$= A \cdot S + B - (K_A + K_B + K\frac{q}{q_{l-1}}) \cdot q_{l-1}$ # where $\frac{q}{q_{l-1}}$ is an integer (the l-th prime element of q_L)
$= A \cdot S + B - K' \cdot q_{l-1}$ # where $K' = K_A + K_B + K\frac{q}{q_{l-1}}$ is an integer
$= A \cdot S + B \bmod q_{l-1}$
$= M + \Delta E$ # since $\Delta M + E < q_0 < q_{l-1}$

As shown above, $(A', B') \bmod q_{l-1}$ decrypts to the same $M + \Delta E$, a plaintext with a scaled error. However, the noise budget (i.e., allowed threshold of the noise) decreases because the ciphertext modulus-to-noise ratio decreases.

CKKS's ModDrop is summarized as follows:

⟨Summary ??⟩ BGV's ModDrop

Given a BGV ciphertext with the l-th multiplicative level $\mathsf{RLWE}_{S,\sigma}(\Delta M) = (A, B) \bmod q_l$, a ModDrop operation is as follows:
$(A', B') \bmod q_{l-1} = (A \bmod q_{l-1}, B \bmod q_{l-1})$

After this, the ciphertext's multiplicative level decreases by 1, the noise's scaling factor Δ and the plaintext are unaffected, and the noise budget (i.e., allowed noise threshold) decreases.

D-4.7 Modulus Switch

- **Reference 1:** Design and implementation of HElib: a homomorphic encryption library [?]
- **Reference 2:** Fully Homomorphic Encryption without Bootstrapping [?]
- **Reference 3:** Homomorphic Evaluation of the AES Circuit [?]

Remember that the requirement of modulus switch is that while we change the ciphertext modulus from q to $\hat{q}$, it should decrypt to the same plaintext M. BGV's modulus switch is similar to that of the RLWE modulus switch (??), but there is an additional requirement, because BGV applies the scaling factor Δ not to plaintext M, but to noise E. In the case of BFV or CKKS, their decryption process only needs to round off the noise in the low-digit area. However, in the case of BGV, the plaintext is in the low-digit area and its decryption process has to remove the noise in the higher-bit area by modulo-t reduction (i.e., the plaintext modulus). More concretely, BGV's modulus switch from $(A, B) \bmod q_l \rightarrow (\hat{A}, \hat{B}) \bmod \hat{q}$ should satisfy the decryption relation such that $((\hat{A} \cdot S + \hat{B}) \bmod \hat{q}) \bmod t = M$. In BGV's modulus switch, $\hat{q}$ does not have to be one of

the multiplicative levels of the ciphertext, and $\hat{q}$ only needs to satisfy the relationship: $\hat{q} < q_l$ and $\hat{q} \equiv 1 \bmod t$. BGV's modulus switch procedure is as follows:

1. The input ciphertext is $\mathsf{ct} = (A, B) \bmod q_l$. We compute new polynomials A' and B' as follows:
   $(A', B') = \left(\left\lceil \frac{\hat{q}}{q_l} \cdot A \right\rfloor, \left\lceil \frac{\hat{q}}{q_l} \cdot B \right\rfloor \right) \pmod{\hat{q}}$

   And we compute the rounding error ϵ_A, ϵ_B as follows:
   $\epsilon_A = \frac{\hat{q}}{q_l} \cdot A - A'$
   $\epsilon_B = \frac{\hat{q}}{q_l} \cdot B - B'$

   , which we rewrite as follows:
   $\hat{q}A = q_l A' + q_l \epsilon_A = q_l A' + \epsilon'_A$ # we denote $\epsilon'_A = q_l \epsilon_A$, where $\epsilon'_A \in \mathbb{Z}_{q_l}$
   $\hat{q}B = q_l B' + q_l \epsilon_B = q_l B' + \epsilon'_B$ # we denote $\epsilon'_B = q_l \epsilon_B$, where $\epsilon'_B \in \mathbb{Z}_{q_l}$

2. We compute new polynomials H_A and H_B as follows:
   $H_A = q_l^{-1} \cdot \epsilon'_A \bmod t$
   $H_B = q_l^{-1} \cdot \epsilon'_B \bmod t$

3. We compute the final mod-switched ciphertext $\hat{\mathsf{ct}}$ as follows:
   $\hat{\mathsf{ct}} = (\hat{A}, \hat{B}) = (A' + H_A,\ B' + H_B) \bmod \hat{q}$

   Note that the computation result of $A' + H_A$ and $B' + H_B$ alone can exceed the range $\mathbb{Z}_{\hat{q}}$, because $A', B' \in \mathbb{Z}_{\hat{q}}$ and $H_A, H_B \in \mathbb{Z}_t$. Therefore, we need to reduce $A' + H_A$ and $B' + H_B$ modulo $\hat{q}$ to derive $\hat{A} \in \mathbb{Z}_{\hat{q}}$ and $\hat{B} \in \mathbb{Z}_{\hat{q}}$.

4. From now on, we will verify that $\hat{\mathsf{ct}}$ is a valid ciphertext satisfying BGV's required decryption relation. First, we can derive the relationship among $\mathsf{ct} = (A, B)$, $A' + H_A$, and $B' + H_B$ as follows:
   $\hat{q} \cdot \mathsf{ct} \bmod t = (\hat{q}A, \hat{q}B) \bmod t$
   $= (q_l A' + \epsilon'_A,\ q_l B' + \epsilon'_B) \bmod t$ # applying step 1's result: $\hat{q}A = q_l A' + \epsilon'_A, \quad \hat{q}B = q_l B' + \epsilon'_B$
   $= (q_l A' + q_l H_A,\ q_l B' + q_l H_B) \bmod t$ # applying step 2's result: $H_A = q_l^{-1} \cdot \epsilon'_A \bmod t, \quad H_B = q_l^{-1} \cdot \epsilon'_B \bmod t$

   $= q_l \cdot (A' + H_A,\ B' + H_B) \bmod t$

   So, $\hat{q} \cdot \mathsf{ct} = q_l \cdot (A' + H_A,\ B' + H_B) \bmod t$. But in BGV, $q_l \equiv q_2 \equiv \cdots q_L \equiv 1 \bmod t$. Thus, the following holds:
   $\mathsf{ct} = (A, B) = (A' + H_A,\ B' + H_B) \pmod t$

5. We can derive the decryption relation of $\hat{\mathsf{ct}}$ from the decryption relation of ct as follows:
   $M = (A \cdot S + B \bmod q_l) \bmod t$ # The BGV decryption relation of $\mathsf{ct} = (A, B) \bmod q_l$
   $= (A \cdot S + B - K \cdot q_l) \bmod t$ # where $K \cdot q_l$ represents the modulo-q_l reduction
   $= ((A' + H_A) \cdot S + B' + H_B - K \cdot q_l) \bmod t$ # applying step 4's result: $A \equiv A' + H_A \bmod t$, $B \equiv B' + H_B \bmod t$

$= ((A' + H_A) \cdot S + B' + H_B - K \cdot \hat{q}) \bmod t$ # since in BGV, $q_1 \equiv q_2 \equiv \cdots q_L \equiv 1 \bmod t$, and we chose $\hat{q}$ such that $\hat{q} \equiv 1 \bmod t$

Now, if we can prove that $(A' + H_A) \cdot S + B' + H_B - K \cdot \hat{q} = (A' + H_A) \cdot S + B' + H_B \bmod \hat{q}$ (i.e., $K \cdot \hat{q}$ reduces $(A' + H_A) \cdot S + B' + H_B$ modulo-$\hat{q}$), then this sufficiently leads to the conclusion that $((A' + H_A) \cdot S + B' + H_B \bmod \hat{q}) \bmod t = M$.

6. The following is also true:
$(A' + H_A) \cdot S + B' + H_B \bmod \hat{q}$
$= |A' + H_A|_{\hat{q}} \cdot S + |B' + H_B|_{\hat{q}} \bmod \hat{q}$
where $|A' + H_A|_{\hat{q}} = A' + H_A \bmod \hat{q}, \quad |B' + H_B|_{\hat{q}} = B' + H_B \bmod \hat{q}$

$= \hat{A} \cdot S + \hat{B} \bmod \hat{q}$ # applying step 3: $\hat{A} = A' + H_A \bmod \hat{q}, \quad \hat{B} = B' + H_B \bmod \hat{q}$

Therefore, proving $(A' + H_A) \cdot S + B' + H_B - K \cdot \hat{q} = (A' + H_A) \cdot S + B' + H_B \bmod \hat{q}$ is equivalent to proving $(A' + H_A) \cdot S + B' + H_B - K \cdot \hat{q} = \hat{A} \cdot S + \hat{B} \bmod \hat{q}$.

7. We will prove that $(A' + H_A) \cdot S + B' + H_B - K \cdot \hat{q} = (A' + H_A) \cdot S + B' + H_B \bmod \hat{q}$ as follows:
$(A' + H_A) \cdot S + B' + H_B - K \cdot \hat{q}$
$= (\frac{\hat{q}}{q_l} \cdot A - \frac{\epsilon'_A}{q_l} + H_A) \cdot S + (\frac{\hat{q}}{q_l} \cdot B - \frac{\epsilon'_B}{q_l} + H_B) - K \cdot \hat{q}$
applying step 1's result: $A' = \frac{\hat{q}}{q_l} \cdot A - \frac{\epsilon'_A}{q_l}, \; B' = \frac{\hat{q}}{q_l} \cdot B - \frac{\epsilon'_B}{q_l}$

$= \left(\frac{\hat{q}}{q_l} \cdot A \cdot S + \frac{\hat{q}}{q_l} \cdot B - K \cdot \hat{q}\right) + H_A \cdot S + H_B - \frac{\epsilon'_A}{q_l} \cdot S - \frac{\epsilon'_B}{q_l}$ # rearranging the terms
$= \frac{\hat{q}}{q_l} \cdot (A \cdot S + B - K \cdot q_l) + H_A \cdot S + H_B - \frac{\epsilon'_A}{q_l} \cdot S - \frac{\epsilon'_B}{q_l}$ # taking out the common factor $\frac{\hat{q}}{q_l}$
$= \frac{\hat{q}}{q_l} \cdot (A \cdot S + B \bmod q_l) + H_A \cdot S + H_B - \frac{\epsilon'_A \cdot S + \epsilon'_B}{q_l}$ # since $A \cdot S + B - K \cdot q_l = A \cdot S + B \bmod q_l$

For successful decryption, every coefficient of the resulting polynomial of the above expression has to be within the range $\mathbb{Z}_{\hat{q}}$ (which means that $K \cdot \hat{q}$ has successfully reduced $(A' + H_A) \cdot S + B' + H_B$ modulo $\hat{q}$). The first term $\frac{\hat{q}}{q_l} \cdot (A \cdot S + B \bmod q_l)$ can be viewed as the original ciphertext ct's noise (with the plaintext message) scaled down by $\frac{\hat{q}}{q_l}$. The coefficients of the second term $H_A \cdot S$ are also small, because $H_A \in \mathbb{Z}_t$ and $S \in \mathbb{Z}_3$. The coefficients of the third term H_B are also small, because $H_B \in \mathbb{Z}_t$. The coefficients of the last term $-\frac{\epsilon'_A \cdot S + \epsilon'_B}{q_l}$ are also small, because $\frac{\epsilon'_A}{q_l}$ and $\frac{\epsilon'_B}{q_l}$ are $\in \mathbb{Z}_{\frac{q_l}{\hat{q}}}$.

Therefore, $(A' + H_A) \cdot S + B' + H_B - K \cdot \hat{q} = (A' + H_A) \cdot S + B' + H_B \bmod \hat{q}$ (provided the above error thresholds hold).

8. Finally, we combine the results of step 6 and 7 as follows:

$(\hat{A} \cdot S + \hat{B}) \bmod \hat{q} \bmod t$
$= ((A' + H_A) \cdot S + B' + H_B \bmod \hat{q}) \bmod t$ # by applying step 6
$= (A \cdot S + B \bmod q_l) \bmod t$ # by applying step 7
$= M$

This means that decrypting $(\hat{A}, \hat{B}) \bmod \hat{q}$ outputs the message M.

We summarize BGV's modulus switch as follows:

⟨Summary ??⟩ BGV's Modulus Switch

Suppose we have the current ciphertext modulus q_l and new ciphertext modulus $\hat{q}$ where $q_l \equiv \hat{q} \equiv 1 \bmod t$ and $\hat{q} < q_l$. Therefore, $\hat{q}$ may or may not be one of the ciphertext moduli comprising a BGV ciphertext's multiplicative level moduli $q_0, q_1, \cdots, q_L$.

BGV's modulus switch from $q_l \to \hat{q}$ is equivalent to updating $(A, B) \bmod q_l$ to $(\hat{A}, \hat{B}) \bmod \hat{q}$ as follows:
$(A', B') = \left(\left\lceil \frac{\hat{q}}{q_l} \cdot A \right\rfloor, \left\lceil \frac{\hat{q}}{q_l} \cdot B \right\rfloor \right) \in \mathcal{R}^2_{\langle n, \hat{q} \rangle}$

$\epsilon'_A = \hat{q} \cdot A - q_l \cdot A'$ # where $\epsilon'_A \in \mathbb{Z}_{q_l}$
$\epsilon'_B = \hat{q} \cdot B - q_l \cdot B'$ # where $\epsilon'_B \in \mathbb{Z}_{q_l}$

$H_A = q_l^{-1} \cdot \epsilon'_A \bmod t$
$H_B = q_l^{-1} \cdot \epsilon'_B \bmod t$

$\hat{\mathsf{ct}} = (\hat{A}, \hat{B}) = (A' + H_A, B' + H_B) \bmod \hat{q}$

After the modulus switch (i.e., the noise scaling factor), $\Delta = t$ stays the same as before. The secret key S also stays the same as before. The noise gets scaled down roughly by $\frac{\hat{q}}{q_l}$, but this does not decrease the noise-to-ciphertext modulus ratio.

D-4.7.1 Difference between Modulus Switch and ModDrop

In the case of CKKS (**??**), the difference between modulus switch and ModDrop is that the former scales down the plaintext's scaling factor by $\frac{q_l}{q_{l-1}} \approx \frac{1}{\Delta}$, whereas ModDrop does not affect the plaintext's scaling factor.

Similarly, in the case of BGV, modulus switch (rescaling) and ModDrop from $q_l \to q_{l-1}$ both lower a BGV ciphertext's modulus from $q_l \to q_{l-1}$. However, the key difference is that rescaling also decreases the noise's scaling factor by $\frac{q_l}{q_{l-1}} \approx \frac{1}{\Delta}$, whereas ModDrop keeps the noise's scaling factor the same as it is. Therefore, rescaling is used only during ciphertext-to-ciphertext multiplication (to be explained in **??**) when scaling down the plaintext's scaling factor in the intermediate ciphertext from $\Delta^2 \to \Delta$. Meanwhile, ModDrop is used to reduce the modulo computation time during an application's routine when it becomes certain that the ciphertext will not undergo any additional

ciphertext-to-ciphertext multiplication (i.e., no need to further decrease the ciphertext's modulus).

The main difference in modulus switch between CKKS and BGV is that the former decreases the plaintext's scaling factor by approximately $\frac{1}{\Delta}$, whereas the latter decreases the noise's scaling factor by approximately $\frac{1}{\Delta}$.

Source Code: Examples of BGV modulus switch can be executed by running this Python script.

D-4.8 Ciphertext-to-Ciphertext Multiplication

- Reference: Introduction to the BGV encryption scheme

Since BGV uses a leveled ciphertext modulus chain like CKKS, BGV's ciphertext-to-ciphertext multiplication scheme is exactly the same as CKKS's scheme (Summary **??** in **??**), except for the rescaling step which uses BGV's modulus switch (**??**).

⟨Summary ??⟩ BGV Ciphertext-to-Ciphertext Multiplication

Suppose we have the following two RLWE ciphertexts:
$\mathsf{RLWE}_{S,\sigma}(M^{\langle 1\rangle} + \Delta E^{\langle 1\rangle}) = (A^{\langle 1\rangle}, B^{\langle 1\rangle})$, where $B^{\langle 1\rangle} = -A \cdot S + M^{\langle 1\rangle} + \Delta E^{\langle 1\rangle}$
$\mathsf{RLWE}_{S,\sigma}(M^{\langle 2\rangle} + \Delta E^{\langle 2\rangle}) = (A^{\langle 2\rangle}, B^{\langle 2\rangle})$, where $B^{\langle 2\rangle} = -A \cdot S + M^{\langle 2\rangle} + \Delta E^{\langle 2\rangle}$

Multiplication between these two ciphertexts is performed as follows:

1. **Basic Multiplication**
   Compute the following:

   $D_0 = B^{\langle 1\rangle} \cdot B^{\langle 2\rangle}$
   $D_1 = A^{\langle 1\rangle} \cdot B^{\langle 2\rangle} + A^{\langle 2\rangle} \cdot B^{\langle 1\rangle}$
   $D_2 = A^{\langle 1\rangle} \cdot A^{\langle 2\rangle}$

   , where $M^{\langle 1\rangle} M^{\langle 2\rangle} + \Delta \cdot (M^{\langle 1\rangle} E^{\langle 2\rangle} + M^{\langle 2\rangle} E^{\langle 1\rangle}) + \Delta^2 E^{\langle 1\rangle} E^{\langle 2\rangle}$
   $= \underbrace{B^{\langle 1\rangle} \cdot B^{\langle 2\rangle}}_{D_0} + \underbrace{(B^{\langle 2\rangle} \cdot A^{\langle 1\rangle} + B^{\langle 1\rangle} \cdot A^{\langle 2\rangle})}_{D_1} \cdot S + \underbrace{(A^{\langle 1\rangle} \cdot A^{\langle 2\rangle})}_{D_2} \cdot \underbrace{S \cdot S}_{S^2}$
   $= D_0 + D_1 \cdot S + D_2 \cdot S^2$

2. **Relinearization**
   $\mathsf{RLWE}_{S,\sigma}(M^{\langle 1\rangle} M^{\langle 2\rangle} + \Delta \cdot (M^{\langle 1\rangle} E^{\langle 2\rangle} + M^{\langle 2\rangle} E^{\langle 1\rangle}) + \Delta^2 E^{\langle 1\rangle} E^{\langle 2\rangle})$
   $= \mathsf{RLWE}_{S,\sigma}(\ D_0 + D_1 \cdot S + D_2 \cdot S^2\ )$
   $\approx C_\alpha + C_\beta$, where $C_\alpha = (D_1, D_0)$, $C_\beta = \langle\ \mathsf{Decomp}^{\beta,l}(D_2), \mathsf{RLev}^{\beta,l}_{S,\sigma}(S^2)\ \rangle$

3. **(Optional) Rescaling**
   To suppress the noise's growing scaling factor from Δ^2 to Δ, switch the ciphertext's modulo from $q \to \hat{q}$ by updating (A, B) to $(\hat{A}, \hat{B})$ according to BGV's modulus switch explained in Summary **??** (**??**).

   After the above update of (A, B) to $(\hat{A}, \hat{B})$, the noise scaling factor $\Delta = t$ and the plaintext

M stay the same, as we proved in **??** that $((\hat{A} \cdot S + B) \bmod \hat{q}) \bmod t = M$.

Swapping the Order of Relinearization and Rescaling: The order of relinearization and rescaling is interchangeable. Running rescaling before relinearization reduces the size of the ciphertext modulus, and therefore the subsequent relinearization can be executed faster.

Details of the Optional Rescaling: Before rescaling, the contents of the ciphertext are $M^{\langle 1 \rangle} M^{\langle 2 \rangle} + \Delta \cdot (M^{\langle 1 \rangle} E^{\langle 2 \rangle} + M^{\langle 2 \rangle} E^{\langle 1 \rangle}) + \Delta^2 E^{\langle 1 \rangle} E^{\langle 2 \rangle} + \epsilon$, where ϵ is a relinearization error. Therefore, after each ciphertext-to-ciphertext multiplication, the noise's scaling factor will become squared as $\Delta^2, \Delta^4, \Delta^8, \cdots$. To reduce such exponential noise growth rate, we can optionally rescale down the ciphertext by $w_l = \dfrac{q_l}{q_{l-1}} > \Delta$ at the end of each relinearization at multiplicative level l, which is the noise's growth rate (effectively keeping the noise scaling factor as Δ). After rescaling, the ciphertext gets scaled down by w_l and then added by a new noise ϵ_2, which generates a new noise ϵ_2. Before the rescaling, the noise grew roughly by the factor of $\Delta = t$ (as the largest noise term is $\Delta^2 E^{\langle 1 \rangle} E^{\langle 2 \rangle}$), but the rescaling process reduces this growth rate by the factor of w_l and then introduces a new constant noise ϵ_2. Therefore, if w_l is sufficiently bigger than $\Delta = t$, the resulting noise will decrease compared to both $\Delta E^{\langle 1 \rangle}$ and $\Delta E^{\langle 2 \rangle}$. Due to this reason, when we design the modulus chain of BGV, we require each w_l to be sufficiently bigger than $\Delta = t$ to effectively reduce the noise growth rate upon each ciphertext-to-ciphertext multiplication (while ensuring the property that its reduction modulo t gives the plaintext M as explained in **??**). Meanwhile, the constant noise term ϵ_2 gets newly added upon each rescaling, but this term becomes part of the rescaled ciphertext, which will be later reduced by the factor of w_{l-1} in the future rescaling. Therefore, BGV's rescaling upon ciphertext-to-ciphertext multiplication effectively suppresses the noise growth.

On the other hand, the above design strategy of noise reduction is inapplicable to CKKS, because in CKKS, we use the scaling factor Δ to scale the message M (not the noise E), and thus CKKS requires each $w_l \approx \Delta$ in order to preserve the plaintext's scaling factor Δ as the same value across ciphertext-to-ciphertext multiplications. Because of this difference in design, CKKS inevitably increases the noise after each ciphertext-to-ciphertext multiplication.

D-4.9 Homomorphic Key Switching

BGV's homomorphic key switching scheme changes an RLWE ciphertext's secret key from S to S'. This scheme is exactly the same as BFV's key switching scheme (Summary **??** in **??**).

⟨Summary ??⟩ BGV's Key Switching

$$\mathsf{RLWE}_{S',\sigma}(M + \Delta E) = (0, B) + \langle \mathsf{Decomp}^{\beta,l}(A),\ \mathsf{RLev}^{\beta,l}_{S',\sigma}(S) \rangle$$

D-4.10 Homomorphic Rotation of Input Vector Slots

BGV's homomorphic rotation scheme of input vector slots is exactly the same as BFV's rotation scheme (Summary **??** in **??**).

⟨Summary ??⟩ BGV's Homomorphic Rotation of Input Vector Slots

Suppose we have a BGV ciphertext and a key-switching key as follows:

$$\mathsf{RLWE}_{S,\sigma}(M + \Delta E) = (A, B), \quad \mathsf{RLev}^{\beta,l}_{S,\sigma}(S^{J(h)})$$

Then, the procedure of rotating the first-half elements of the ciphertext's original input vector $\vec{v}$ by h positions to the left (in a wrapping manner among them) and the second-half elements of $\vec{v}$ by h positions to the left (in a wrapping manner among them) is as follows:

1. Update $A(X)$, $B(X)$ to $A(X^{J(h)})$, $B(X^{J(h)})$.
2. Perform the following key switching (**??**) from $S(X^{J(h)})$ to $S(X)$:
   $\mathsf{RLWE}_{S(X),\sigma}(M(X^{J(h)}) + \Delta E(X^{J(h)}))$
   $= (0, B(X^{J(h)})) \; + \; \langle \mathsf{Decomp}^{\beta,l}(A(X^{J(h)})), \; \mathsf{RLev}^{\beta,l}_{S(X),\sigma}(S(X^{J(h)}))\rangle$

D-4.11 Modulus Bootstrapping

- Reference: Bootstrapping for BGV and BFV Revisited [**?**]

BGV's bootstrapping shares some common aspects with both BFV and CKKS's bootstrapping. The goal of BGV's bootstrapping is the same as that of CKKS, but the internal technique is closer to that of BFV. Like CKKS, BGV's bootstrapping resets the depleted ciphertext modulus from $q_l \to q_L$ (strictly speaking, from $q_l \to q_{l'}$ such that $l < l' < L$ because the bootstrapping operation itself consumes some multiplicative levels). This modulus transition effectively not only resets the multiplicative level but also reduces the noise-to-ciphertext modulus ratio. To achieve this goal, one might think that BGV's bootstrapping can take the same ModRaise approach used by CKKS's bootstrapping. However, this is not a directly applicable solution, because CKKS uses the sine approximation technique to eliminate the q_0-overflows after the mod-raise. On the other hand, BGV is an exact encryption scheme which does not allow approximation of plaintext values. Therefore, BGV uses BFV's digit extraction approach to eliminate its modulus overflows. To use digit extraction, like in the case of BFV, BGV also has to modify the plaintext modulus to a specially prepared one, p^{ε}. To configure both the plaintext modulus and the ciphertext modulus to desired values (i.e., p^{ε} and q_L), BGV uses the homomorphic decryption technique like BFV.

The technical details of BGV's bootstrapping are as follows.

Suppose that we have an RLWE ciphertext $(A, B) = \mathsf{RLWE}_{S,\sigma}(M) \bmod q_l$, where $A \cdot S + B = M + \Delta E$, $\Delta = t = p$ (a prime), and q_l is the ciphertext modulus of the current multiplicative level.

1. **<u>Modulus Switch</u> from $q_l \to \hat{q}$:** BFV's bootstrapping initially switches the ciphertext modulus from $q \to p^{\varepsilon-1}$ where $q \gg p^{\varepsilon} > t = p$. On the other hand, BGV's bootstrapping switches the ciphertext modulus to $\hat{q}$ that is a special modulus satisfying the relation: $\hat{q} \equiv 1 \bmod p^{\varepsilon}$ and $\hat{q} > p^{\varepsilon}$ (where p^{ε} will be explained in the next step). In order for a modulus switch from $q_l \to \hat{q}$ (i.e., the special modulus) to be possible, the prime factor(s) comprising $\hat{q}$ have to be congruent with $q_0, \cdots, q_L \bmod t$, so that we can do a modulus switch from $q_l \cdot \hat{q} \to \hat{q}$ (based on the technique learned in **??**). Eventually, this step's modulus switch transforms the ciphertext $(A, B) \bmod q_l$ to $(\hat{A}, \hat{B}) \bmod \hat{q}$, during which the plaintext modulus (i.e., noise's scaling factor) stays the same.

2. **Ciphertext Coefficient Multiplication by $p^{\varepsilon-1}$:** The constant $p^{\varepsilon-1}$ is multiplied to each coefficient of the ciphertext polynomials, updating the ciphertext to $p^{\varepsilon-1} \cdot (\hat{A}, \hat{B}) = (A', B') \bmod \hat{q}$, where $A' = p^{\varepsilon-1}\hat{A}$ and $B' = p^{\varepsilon-1}\hat{B}$. This operation updates the original decryption relation $\hat{A} \cdot S + \hat{B} = M + pE + K\hat{q}$ to $A' \cdot S + B' = p^{\varepsilon-1}M + p^{\varepsilon}E + K'\hat{q}$ (where $K' = K \cdot p^{\varepsilon-1}$). Notice that the plaintext modulus (i.e., noise's scaling factor) has been changed from $p \to p^{\varepsilon}$. When choosing ε, BGV enforces the following additional constraint: $\hat{q} > p^{\varepsilon}$ and $\hat{q} \equiv 1 \bmod p^{\varepsilon}$.

3. **ModRaise:** We mod-raise $(\hat{A}, \hat{B}) \bmod \hat{q}$ to $(\hat{A}, \hat{B}) \bmod q_L$, where $\hat{q} \ll q_L$. The mod-raised ciphertext's decryption relation is as follows:
$\hat{A} \cdot S + \hat{B} = p^{\varepsilon-1}M + p^{\varepsilon}E + K'\hat{q} \bmod q_L$

   Note that $K'\hat{q}$ is the $\hat{q}$-multiple overflow and does not get reduced modulo q_L, because $K'\hat{q} \ll q_L$. We saw the same situation in the CKKS bootstrapping's ModRaise (**??**) which resets the ciphertext modulus from $q_0 \to q_L$ at the cost of incurring a Kq_0 overflow, which is to be removed by EvalExp's homomorphic (approximate) sine graph evaluation (**??**). Likewise, BGV's mod-raised ciphertext $(\hat{A}, \hat{B}) \bmod q_L$ is $\mathsf{RLWE}_{S,\sigma}(p^{\varepsilon-1}M + p^{\varepsilon}E + K'\hat{q}) \bmod q_L$, an encryption of $p^{\varepsilon-1}M + p^{\varepsilon}E + K'\hat{q}$. In the later step, we will use digit extraction to homomorphically eliminate $K'\hat{q}$ like we did in BFV's bootstrapping. The reason BGV's bootstrapping uses digit extraction instead of approximated sine evaluation is that BGV is an exact encryption scheme like BFV (not an approximate scheme like CKKS).

4. **CoeffToSlot:** This step works the same way as CKKS and BFV's CoeffToSlot step: move the coefficients of polynomial $p^{\varepsilon-1}M + p^{\varepsilon}E + \hat{q}K'$ to the input vector slots of a new ciphertext. We denote polynomial $Z = p^{\varepsilon-1}M + p^{\varepsilon}E + \hat{q}K'$, and each i-th coefficient of Z as z_i. For the CoeffToSlot step, we homomorphically compute $Z \cdot n^{-1} \cdot \tilde{W} \cdot I_n^R$. Then, each input vector slot of the resulting ciphertext ends up storing each z_i of the polynomial Z.

5. **Digit Extraction:** At this point, each input vector slot contains each coefficient of $p^{\varepsilon-1}M + p^{\varepsilon}E + \hat{q}K'$, which is $p^{\varepsilon-1}m_i + p^{\varepsilon}e_i + \hat{q}k'_i$. Recall that we designed the lowest multiplicative level's ciphertext modulus $\hat{q}$ and the homomorphic multiplication factor p^{ε} such that $\hat{q} \equiv 1 \bmod p^{\varepsilon}$, or $\hat{q} = k^{\langle \hat{q} \rangle} \cdot p^{\varepsilon} + 1$ for some positive integer $k^{\langle \hat{q} \rangle}$. Therefore, the following holds:
$p^{\varepsilon-1}m_i + p^{\varepsilon}e_i + \hat{q}k'_i$
$= p^{\varepsilon-1}m_i + p^{\varepsilon}e_i + k'_i \cdot (k^{\langle \hat{q} \rangle} \cdot p^{\varepsilon} + 1)$ # applying $\hat{q} = k^{\langle \hat{q} \rangle} \cdot p^{\varepsilon} + 1$
$= p^{\varepsilon-1}m_i + k'_i + p^{\varepsilon} \cdot (e_i + k'_i \cdot k^{\langle \hat{q} \rangle})$ # rearranging the terms
$= p^{\varepsilon-1}m_i + k'_i + p^{\varepsilon} \cdot k^{\langle \hat{q}+\varepsilon \rangle}$ # where $k^{\langle \hat{q}+\varepsilon \rangle} = e_i + k'_i \cdot k^{\langle \hat{q} \rangle}$
$= p^{\varepsilon-1}m_i + k'_i \bmod p^{\varepsilon}$

   To eliminate k'_i from the above, we will use the same digit extraction polynomial $G_{\varepsilon,v}$ as in BFV (**??**):
$z_i = d_0 + \left(\sum_{j=\varepsilon'}^{\varepsilon-1} d_* p^j \right)$ # where $d_0 \in \mathbb{Z}_p$, and d_* can be any integer, and $1 \leq \varepsilon' \leq w$
$F_{\varepsilon}(z_i) \equiv d_0 \bmod p^{\varepsilon'+1}$
$G_{\varepsilon,v}(z_i) \equiv z_i - \underbrace{F_{\varepsilon} \circ F_{\varepsilon} \circ \cdots \circ F_{\varepsilon}}_{v \text{ times}}(z_i) \bmod p^{\varepsilon}$

We evaluate the digit extraction polynomial $G_{\varepsilon,v}$ for $v = \{\varepsilon-1, \varepsilon-2, \cdots, 1\}$ recursively total $\varepsilon - 1$ times, at each coefficient z_i of polynomial Z stored at input vector slots. This operation finally zeros out the least significant (base-p) $\varepsilon - 1$ digits of z_i as follows:
$G_{\varepsilon,1} \circ G_{\varepsilon,2} \circ \cdots \circ G_{\varepsilon,\varepsilon-2} \circ G_{\varepsilon,\varepsilon-1}(z_i) \bmod p^{\varepsilon}$
$= p^{\varepsilon-1} m_i \bmod p^{\varepsilon}$
$= p^{\varepsilon-1} m_i + k_i'' p^{\varepsilon}$

, where $k_i'' p^{\varepsilon}$ is some multiple of p^{ε} to account for the original p^{ε}-overflow term plus an additional p^{ε}-overflows generated during the digit extraction. Note that the digit extraction step reduces the ciphertext modulus from $q_L \to q_{l'}$ (where l' is an integer smaller than L), because the homomorphic evaluation of the polynomial $G_{\varepsilon,v}$ requires some ciphertext-to-ciphertext multiplications, which consume some multiplicative levels.

6. **Homomorphic Multiplication by $p^{-(\varepsilon-1)}$:** The output of the digit extraction step is $p^{\varepsilon-1} m_i + k_i'' p^{\varepsilon}$ stored in each input vector slot. We homomorphically multiply $|p^{-(\varepsilon-1)}|_{p^{\varepsilon}}$ to it, which is a modulo-p^{ε} inverse of $p^{\varepsilon-1}$. Note that $p^{-(\varepsilon-1)}$ is guaranteed to exist because $\mathbb{Z}_{p^{\varepsilon}}$ is a finite field (i.e., Galois field) whose every element is guaranteed to have its counterpart inverse (Theorem **??** in **??**). Multiplying $p^{-(\varepsilon-1)}$ to $p^{\varepsilon-1} m_i + k'' i p^{\varepsilon}$ is equivalent to an exact division of $p^{\varepsilon-1} m_i + k'' i p^{\varepsilon}$ by $p^{\varepsilon-1}$, because $p^{\varepsilon-1} m_i + k'' i p^{\varepsilon}$ is exactly divisible by $p^{\varepsilon-1}$. We homomorphically compute the following:
$|p^{-(\varepsilon-1)}|_{p^{\varepsilon}} \cdot (p^{\varepsilon-1} m_i + k_i'' p^{\varepsilon}) = m_i + k_i'' p \pmod{p^{\varepsilon}}$

   Note that the plaintext value $m_i + k_i'' p \bmod p^{\varepsilon}$ is also equivalent to $m_i + k_i'' p \bmod p$ (because p divides p^{ε}), and is also equivalent to $m_i \bmod q$ (i.e., the message portion without the noise is m_i). Therefore, homomorphically multiplying $|p^{-(\varepsilon-1)}|_{p^{\varepsilon}}$ to a ciphertext that encrypts $p^{\varepsilon-1} m_i + k_i'' p^{\varepsilon}$ is equivalent to switching the plaintext modulus from $p^{\varepsilon} \to p$.

   In an alternative design, one can eliminate this step of homomorphic multiplication by $p^{-(\varepsilon-1)}$ by re-designing the digit extraction algorithm to gradually shift down the digits by total (base-p) $\varepsilon - 1$ digits (by multiplying by $|p^{-1}|_{p^{\varepsilon}}$ at the end of each round of digit extraction).

7. **SlotToCoeff:** This step works the same way as BFV's SlotToCoeff step: move $m_i + k_i'' p$ stored in the input vector slots back to the polynomial coefficient positions by homomorphically multiplying with $\tilde{W}^*$.
   Meanwhile, the coefficient domain is in modulo q_L. From modulo-q_L's perspective, the result of step 5 is $|p^{-(\varepsilon-1)}|_{p^{\varepsilon}} \cdot (p^{\varepsilon-1} m_i + k_i'' p^{\varepsilon}) \bmod q_L$. It is guaranteed that $|p^{-(\varepsilon-1)}|_{p^{\varepsilon}} \cdot (p^{\varepsilon-1} m_i + k_i'' p^{\varepsilon}) < q_L$, because $p^{\varepsilon} \ll q_L$. Therefore, result of SlotToCoeff is a set of polynomial coefficients whose noise is within the noise budget of q_L.

8. **Noise Term Re-interpretation:** The output of the SlotToCoeff step is $\mathsf{RLWE}_{S,\sigma}(M + K''p) \bmod q_{l'}$, which also contains some noise term $E'p$ generated during the homomorphic operations of step $2 \sim 6$. Therefore, we can view the $K''p$ term in the plaintext as part of the noise of the ciphertext. In other words, we can view $\mathsf{RLWE}_{S,\sigma}(M + K''p)$ with some noise term $E'p$ as a ciphertext $\mathsf{RLWE}_{S,\sigma}(M)$ with the noise term $E'p + K''p = (E' + K'') \cdot p$. This step does not require any additional computation. The size of the coefficients of K'' is upper-bounded because

the operations of the CoeffToSlot, digit extraction, and SlotToCoeff steps are fixed. With a proper setup of the cryptographic parameters of BGV, we can guarantee that the noise-to-ciphertext modulus ratio always gets decreased after BGV's bootstrapping (i.e., $\frac{||E+K'||_\infty}{q_L} < \frac{||E'+K''||_\infty}{q_{l'}} < \frac{||E||_\infty}{\hat{q}}$, where $\hat{q} < q_l < q_{l'} < q_L$, and $||P||_\infty$ denotes the maximum absolute coefficient value of polynomial P).

D-4.11.1 The Reason for Modulus Switch from $q_l \to \hat{q}$

BGV switches the modulus from $q_l \to \hat{q}$ to eliminate the q_l-multiple overflows during bootstrapping. After switching the modulus $q_l \to \hat{q}$ and then ModRaise, the encrypted plaintext gets the $K'\hat{q}$ overflow term, which can be reduced to K' from the plaintext modulus's perspective due to the special property $\hat{q} \equiv 1 \bmod p^\varepsilon$ (where p^ε is the plaintext modulus).

D-4.11.2 ModRaise instead of Homomorphic Decryption

In the case of BFV's bootstrapping, we need homomorphic decryption (**??**) because we need to simultaneously change the ciphertext's plaintext scaling factor from $p^{\varepsilon-1} \to \left\lfloor \frac{q}{p} \right\rfloor$ and the ciphertext modulus from $p^\varepsilon \to q$. On the other hand, in the case of CKKS's bootstrapping, ModRaise instead of homomorphic decryption is sufficient because we only need to change the ciphertext modulus from $q_0 \to q_L$ while keeping the same plaintext scaling factor $\Delta \approx \frac{q_l}{q_{l-1}}$. Similarly, in the case of BGV's bootstrapping, ModRaise instead of homomorphic decryption is sufficient because we only need to change the ciphertext modulus from $\hat{q} \to q_L$ while keeping the noise scaling factor (i.e., the plaintext modulus) $\Delta = p^\varepsilon$.

D-4.11.3 The Choice of ε

The larger ε is, the greater the (base-p) digit-wise gap between $p^{\varepsilon-1}M$ and K' becomes, and thus the less likely it is that the decryption would fail (i.e., fail to zero out K'). But a larger ε means the digit extraction operation would be more expensive.

D-4.11.4 Generalization to $\Delta = p^r$

Like the case of BFV's bootstrapping (Summary **??** in **??**), we can generalize the plaintext modulus (i.e., noise scaling factor) to p^r where p is a prime and r can be any positive integer.

D-5 RNS-variant FHE Schemes

FHE parameters of BFV, BGV, or CKKS schemes which are secure enough sometimes require the ring size of polynomial coefficients to be 1000 bits or more, which consumes much computational resources for 64-bit CPU architectures. To make the computation efficient, we can alternatively represent the coefficients of ciphertext polynomials by the number residue system (RNS) **??**, which allows modulo addition and multiplication of elements from a large ring (e.g., 1000 bits) by the combination of values computed in small rings (e.g., 32∼64 bits), each of which compactly fits in 64-bit CPU registers. Modern BFV, BGV, and CKKS schemes adopt this RNS approach by default for efficient computation of large values. These are called RNS-variant FHE schemes.

While RNS can directly compute modulo addition and multiplication, it does not directly support other operations such as ModRaise or modulus switch, which are essential operations for all FHE schemes. This section explains how we can design such corner-case operations based on RNS to accomplish a complete design of RNS-based FHE schemes. Besides BFV, BGV, and CKKS, TFHE can also theoretically use RNS for representing its ciphertext coefficients. However, TFHE's practically used coefficient size is less than 2^{32} (or 2^{64}), which compactly fits in 32-bit (or 64-bit) modern CPU registers. Therefore, TFHE does not need RNS. Thus, this section will focus on RNS-based operations for BFV, BGV, and CKKS.

Particularly in this section, we assume the modulo reduction $a \bmod q = |a|_q$ implicitly uses a centered (i.e., signed) residue representation (**??**) whose modulo overflow & underflow boundaries are $\frac{q}{2} - 1$ and $-\frac{q}{2}$, respectively. This assumption is necessary to eliminate a certain modulo reduction operation when designing FastBconvEx (**??**)– by using the assumption of limiting the possible range of certain residue arithmetic as discussed in **??**.

Required Background

- ??: ??
- ??: ??
- ??: ??
- ??: ??
- ??: ??
- ??: ??
- ??: ??
- ??: ??
- ??: ??
- ??: ??
- ??: ??
- ??: ??
- ??: ??
- ??: ??
- ??: ??
- ??: ??
- ??: ??
- ??: ??
- ??: ??

- ??: ??

D-5.1 Fast Base Conversion: FastBConv

- **Reference 1:** A Full RNS Variant of FV-like Somewhat Homomorphic Encryption Schemes [?]
- **Reference 2:** BASALISC: Programmable Hardware Accelerator for BGV FHE [?]

Suppose we have $x \in \mathbb{Z}_q$ (where q is a big modulus). Then, we can express x by using RNS (??) as $(x_1, x_2, \cdots, x_k)$, where each $x_i \in \mathbb{Z}_{q_i}$, $\prod_{i=1}^{k} q_i = q$, and $\{q_1, q_2, \cdots, q_k\}$ are co-prime. In RNS, we define base conversion as an operation of converting the RNS residues $(x_1, x_2, \cdots, x_k) \in \mathbb{Z}_{q_1} \times \mathbb{Z}_{q_2} \times \cdots \times \mathbb{Z}_{q_k}$ into $(c_1, c_2, \cdots, c_k) \in \mathbb{Z}_{b_1} \times \mathbb{Z}_{b_2} \times \cdots \times \mathbb{Z}_{b_l}$, where $\{b_1, b_2, \cdots, b_l\}$ are a new base, and $\{q_1, q_2, \cdots, q_k\}$ and $\{b_1, b_2, \cdots, b_l\}$ are all co-prime. The relationship between x and c is: $c = |x|_b$ (where $x \in \mathbb{Z}_q$ and $c \in \mathbb{Z}_b$). The standard way of performing base conversion is assembling $(x_1, x_2, \cdots, x_k)$ into x by computing $x = \sum_{i=1}^{k} |x_i z_i|_{q_i} \cdot y_i \bmod q$ (where $y_i = \frac{q}{q_i}$ and $z_i = y_i^{-1} \bmod q_i$), and then computing $c_j \equiv x \bmod b_j$ for $j \in [1, l]$. However, this computation is slow if the modulus q is large. To compute the base conversion *fast*, we design the fast base conversion operation FastBConv as follows:

⟨Summary ??⟩ Fast Base Conversion: FastBConv

Input: $(x_1, x_2, \cdots, x_k) \in \mathbb{Z}_{q_1} \times \mathbb{Z}_{q_2} \times \cdots \times \mathbb{Z}_{q_k}$ # which represents the big value $x \in \mathbb{Z}_q$, where $q = \prod_{i=1}^{k} q_i$, and $\{q_1, q_2, \cdots, q_k\}$ are co-prime

$$\mathsf{FastBConv}(x, q, b) = \mathsf{FastBConv}(\{x_i\}_{i=1}^{k}, \{q_i\}_{i=1}^{k}, \{b_i\}_{i=1}^{l})$$
$$= \left(\sum_{i=1}^{k} |x_i \cdot z_i|_{q_i} \cdot y_i \bmod b_j \right)_{j \in [1,l]}$$

where $y_i = \frac{q}{q_i}$, $z_i = y_i^{-1} \bmod q_i$, and $b = \prod_{i=1}^{l} b_i$

$= (c_1, c_2, \cdots, c_l) \in \mathbb{Z}_{b_1} \times \mathbb{Z}_{b_2} \times \cdots \times \mathbb{Z}_{b_l}$ # which represents the big value $c \in \mathbb{Z}_b$

The input to this FastBConv function is a list of RNS residues $(x_1, x_2, \cdots, x_k)$ having the prime moduli $(q_1, q_2, \cdots, q_k)$ as the base. This RNS vector represents the big value:
$x = \left(\sum_{i=1}^{k} x_i \cdot y_i \cdot z_i \right) \bmod q = \left(\sum_{i=1}^{k} |x_i \cdot z_i|_{q_i} \cdot y_i \right) \bmod q$ # Theorem ??.1

The output of this FastBConv function is a list of RNS residues $(c_1, c_2, \cdots, c_l)$ having the prime moduli $(b_1, b_2, \cdots, b_l)$ as the base. This RNS vector represents the big value
$c = \left(\sum_{i=1}^{l} c_i \cdot y'_i \cdot z'_i \right) \bmod b = \left(\sum_{i=1}^{l} |c_i \cdot z'_i|_{b_i} \cdot y'_i \right) \bmod b$ # where $y'_i = \frac{b}{b_i}$ and $z'_i = y_i'^{-1} \bmod b_i$

The relationship between c and x is as follows:
$c = x + uq \bmod b$ (where u is an integer with the magnitude $|u| \leq \frac{k}{2} + 1$)
i.e. the fast-base-converted c gets noise $|uq|_b$

Proof.

We will prove why a fast base conversion of x into c gets an additional noise $|u \cdot q|_b$ (where integer $|u| \leq \frac{k}{2}+1$) compared to a standard base conversion. If we did a standard (i.e., exact) base conversion of x from base moduli $(q_1, \cdots, q_k)$ to $(b_1, \cdots, b_l)$, then we would compute the following:

$$\left(\left(\sum_{i=1}^{k} |x_i \cdot z_i|_{q_i} \cdot y_i \bmod q\right) \bmod b_j\right)_{j\in[1,l]} = (x \bmod b_j)_{j\in[1,l]}$$

But FastBConv omits the intermediate (big) reduction modulo q and directly applies (small) reduction modulo b_j for the sake of fast computation, so that our conversion process does not need to handle large values whose magnitude can be as large as $\pm\frac{q}{2}$. In this approach of fast base conversion, for each $i \in [1,k]$, the computation result of $|x_i \cdot z_i|_{q_i} \cdot y_i$ is some value between $\left[-\left\lceil\frac{q}{2}\right\rceil, \left\lfloor\frac{q}{2}\right\rfloor\right]$, because $|x_i \cdot z_i|_{q_i}$ is some integer between $\left[-\left\lceil\frac{q_i}{2}\right\rceil, \left\lfloor\frac{q_i}{2}\right\rfloor\right]$ and $y_i = \frac{q}{q_i}$. Therefore, $-\frac{q+1}{2} \leq |x_i \cdot z_i|_{q_i} \cdot y_i \leq \frac{q}{2}$. If we sum k such values for $i \in [1,k]$, then the total sum $x' = \sum_{i=1}^{k} |x_i \cdot z_i|_{q_i} \cdot y_i = x + u \cdot q$ (Summary **??** in **??**) for some integer u (where $u \cdot q$ represents the q-multiple overflows). And since we have shown that $-\frac{q+1}{2} \leq |x_i \cdot z_i|_{q_i} \cdot y_i \leq \frac{q}{2}$ for each $i \in [1,k]$, uq has to be greater than $-k \cdot \frac{q+1}{2}$ and smaller than $k \cdot \frac{q}{2}$ (i.e., u is an integer between $-\frac{k}{2} - 1 \leq u \leq \frac{k}{2}$). Therefore, $\sum_{i=1}^{k} |x_i \cdot z_i|_{q_i} \cdot y_i$ can have maximum $-\left(\frac{k}{2}+1\right) \cdot q$ underflows and $\frac{k}{2} \cdot q$ overflows. Thus, while standard (i.e., exact) base conversion computes each residue as $\hat{c}_j = \left(\sum_{i=1}^{k} |x_i \cdot z_i|_{q_i} \cdot y_i \bmod q\right) \bmod b_j$ (i.e., $\hat{c}_j = x \bmod b_j$), fast (i.e., approximate) base conversion computes each residue as $c_j = \left(\sum_{i=1}^{k} |x_i \cdot z_i|_{q_i} \cdot y_i\right) \bmod b_j$ (i.e., $c_j = x + uq \bmod b_j$, where integer $|u| \leq \frac{k}{2}+1$). Notice that the residual difference (i.e., error) between each $\hat{c}_j$ and c_j is $uq \bmod b_j$, and the collective noise generated by fast base conversion from $q \to b$ is $uq \bmod b$. Also, note that the RNS residue vector $(c_1, c_2, \cdots, c_l) \in \mathbb{Z}_{b_1} \times \mathbb{Z}_{b_2} \times \cdots \times \mathbb{Z}_{b_l}$ represents the big value $c = x + uq \bmod b$.

Importantly, FastBConv does not guarantee the correctness of base conversion, because the q-multiple overflow would generate a non-negligible error. Yet, FastBConv is used as an essential building block for various RNS-based operations such as $\mathsf{ModRaise}_{\mathsf{RNS}}$ (**??**) and $\mathsf{ModSwitch}_{\mathsf{RNS}}$ (**??**). □

D-5.2 RNS-based ModRaise: $\mathsf{ModRaise}_{\mathsf{RNS}}$

- Reference: A Full RNS Variant of Approximate Homomorphic Encryption [**?**]

ModRaise is an operation of raising a ciphertext's modulus from q to qb (where $q \ll qb$). We used ModRaise in BFV's ciphertext-to-ciphertext multiplication (Summary **??** in **??**) and in CKKS's

modulus bootstrapping (Summary ?? in ??). The RNS-based ModRaise operation is designed as follows:

⟨Summary ??⟩ $\mathsf{ModRaise}_{\mathsf{RNS}}$

Input: $(x_1, x_2, \cdots, x_k) \in \mathbb{Z}_{q_1} \times \mathbb{Z}_{q_2} \times \cdots \times \mathbb{Z}_{q_k}$ # which represents the big value $x \in \mathbb{Z}_q$

$\mathsf{ModRaise}_{\mathsf{RNS}}(\{x_i\}_{i=1}^{k}, q, qb)$ # where q and b are co-prime
$= \mathsf{FastBConv}_{\mathsf{RNS}}(\{x_i\}_{i=1}^{k}, q, qb)$
$= (x_1, x_2, \cdots, x_k, \mathsf{FastBConv}(\{x_i\}_{i=1}^{k}, q, b))$
$= (x_1, x_2, \cdots, x_k, c_1, c_2, \cdots, c_l) \in \mathbb{Z}_{q_1} \times \mathbb{Z}_{q_2} \times \cdots \times \mathbb{Z}_{q_k} \times \mathbb{Z}_{b_1} \times \mathbb{Z}_{b_2} \times \cdots \times \mathbb{Z}_{b_l}$
$= (\chi_1, \chi_2, \cdots, \chi_{k+l}) \in \mathbb{Z}_{q_1} \times \mathbb{Z}_{q_2} \times \cdots \times \mathbb{Z}_{q_k} \times \mathbb{Z}_{b_1} \times \mathbb{Z}_{b_2} \times \cdots \times \mathbb{Z}_{b_l}$
which represents the value $\chi \in \mathbb{Z}_{qb}$

The relationship between χ and x is as follows:
$\chi \equiv x + u \cdot q \bmod qb$ # the noise generated by $\mathsf{ModRaise}_{\mathsf{RNS}}$ is $|uq|_{qb}$ (where integer $|u| \leq \frac{k}{2} + 1$)
$\chi \equiv x \bmod q$

Proof.

In ??, we proved that $x' = \sum_{i=1}^{k} |x_i \cdot z_i|_{q_i} \cdot y_i = x + u \cdot q$ (where integer $|u| \leq \frac{k}{2} + 1$). Therefore, the following holds:

$x' \equiv x_i \bmod q_i$ for $i \in [1, k]$ # since $x' = x + u \cdot q \equiv x_i \bmod q_i$ (as q_i divides q, so $x \equiv x_i \bmod q_i$)
$x' \equiv c_j \bmod b_j$ for $j \in [1, l]$ # where each $c_j = x + uq \bmod b_j$

Therefore, $x' \bmod qb$ can be represented as the following RNS residues:
$(x_1, x_2, \cdots, x_k, c_1, c_2, \cdots, c_l) \in \mathbb{Z}_{q_1} \times \mathbb{Z}_{q_2} \times \cdots \times \mathbb{Z}_{q_k} \times \mathbb{Z}_{b_1} \times \mathbb{Z}_{b_2} \times \cdots \times \mathbb{Z}_{b_l}$
$= (x_1, x_2, \cdots, x_k, \mathsf{FastBConv}(\{x_i\}_{i=1}^{k}, q, b)) \in \mathbb{Z}_{q_1} \times \mathbb{Z}_{q_2} \times \cdots \times \mathbb{Z}_{q_k} \times \mathbb{Z}_{b_1} \times \mathbb{Z}_{b_2} \times \cdots \times \mathbb{Z}_{b_l}$

Our ideal goal of mod-raising $x \in \mathbb{Z}_q$ from $q \to qb$ is to derive an RNS vector of $x \bmod qb$. However, the above RNS vector represents $x' \bmod qb$, where $x' = x + uq$ (with integer $|u| \leq \frac{k}{2} + 1$). Therefore, we can interpret the above RNS vector as representing $x \bmod qb$ with the additional noise $|uq|_{qb}$.

□

D-5.3 RNS-based ModDrop: $\mathsf{ModDrop}_{\mathsf{RNS}}$

ModDrop (??, ??) is an operation of decreasing a ciphertext's modulus from $q \to q'$ (where q' divides q) without affecting the plaintext's scaling factor(in the case of CKKS) or the noise's scaling factor (in the case of BGV).

In an RNS-based ciphertext representation, ModDrop is equivalent to removing some of the base moduli in the ciphertext without affecting the scaling factor Δ. This can be achieved by converting the ciphertext's base from q to $\bar{q}$ where the base moduli set of $\bar{q}$ are a subset of that of q; that is, $\bar{q}$ divides q. Specifically, suppose that we have an input $(x_1, x_2, \cdots, x_k) \in \mathbb{Z}_{q_1} \times \mathbb{Z}_{q_2} \times \cdots \times \mathbb{Z}_{q_k}$, and a new subset base $\bar{q} = q_1 \cdot q_2 \cdot \cdots \cdot q_{k'}$, where $k' < k$. In this setup, the fast base conversion from

$q \to \bar{q}$ is equivalent to simply extracting the input value's RNS residues associated with the base moduli $(q_1, q_2, \cdots, q_{k'})$. This is because of the following reasoning:

$$\mathsf{FastBConv}(\{x_i\}_{i=1}^{k}, q, \bar{q}) = \left(\sum_{i=1}^{k} |x_i \cdot z_i|_{q_i} \cdot y_i \bmod q_j \right)_{j \in [1,k']}$$
$$= x + uq \bmod \bar{q} \ \# \text{ Summary ?? in ??}$$
$$= x \bmod \bar{q} \ \# \ uq \text{ gets eliminated because } \bar{q} \text{ divides } uq$$
$$= (x_1, x_2, \cdots, x_{k'}) \in \mathbb{Z}_{q_1} \times \mathbb{Z}_{q_2} \times \cdots \times \mathbb{Z}_{q_{k'}}$$

Notice that the above fast base conversion from $q \to \bar{q}$ (where $\bar{q}$ divides q) does not generate any noise. This is different from the case of fast base conversion from $q \to b$ (Summary **??** in **??**) where q and b are co-prime, which generates the noise $|uq|_b$ (where integer $|u| \leq \frac{k}{2} + 1$).

The ModDrop operation is supported in all of BFV, BGV, and CKKS ciphertexts that are represented in RNS forms. However, note that ModDrop is possible only if the scaled plaintext (in the case of BFV and CKKS) or the scaled noise (in the case of BGV) does not exceed the ciphertext modulus after the mod-drop operation, because otherwise correct decryption is not possible. $\mathsf{ModDrop}_{\mathsf{RNS}}$ is summarized as follows:

⟨Summary ??⟩ $\mathsf{ModDrop}_{\mathsf{RNS}}$

Input: $(x_1, x_2, \cdots, x_k) \in \mathbb{Z}_{q_1} \times \mathbb{Z}_{q_2} \times \cdots \times \mathbb{Z}_{q_k}$

$$\mathsf{FastBConv}(\{x_i\}_{i=1}^{k}, q, \bar{q}) = \left(\sum_{i=1}^{k} |x_i \cdot z_i|_{q_i} \cdot y_i \bmod q_j \right)_{j \in [1,k']}$$
$\#$ where $\bar{q}$ is a product of co-primes $q_1 \cdot q_2 \cdot \cdots \cdot q_{k'}$, and $\bar{q}$ divides q
$= (x_1, x_2, \cdots, x_{k'}) \in \mathbb{Z}_{q_1} \times \mathbb{Z}_{q_2} \times \cdots \times \mathbb{Z}_{q_{k'}}$ $\#$ no noise generated during the conversion

D-5.4 RNS-based Modulus Switch: $\mathsf{ModSwitch}_{\mathsf{RNS}}$

Modulus switch is an operation of reducing a ciphertext's modulus from q to q' (where $q' < q$) and updating the target value from x to $\left\lceil x \cdot \frac{q'}{q} \right\rfloor$. Modulus switch is used for lowering the multiplicative level of a ciphertext upon each ciphertext-to-ciphertext multiplication (in the case of BFV, CKKS, or BGV) or even upon each ciphertext-to-plaintext multiplication (in the case of CKKS). Upon each modulus switch from $q \to q'$ of a ciphertext, the scaling factor of the underlying plaintext in the ciphertext also gets reduced by the same proportion: $\frac{q'}{q}$.

The modulus switch operation of an RNS-based ciphertext is denoted as $\mathsf{ModSwitch}_{\mathsf{RNS}}$, which requires that the output base moduli are a subset of the input base moduli. In other words, like the case of $\mathsf{ModDrop}_{\mathsf{RNS}}$, it only supports a modulus switch from $qb \to q$, where q and b are co-prime.

Suppose we have $(\chi_1, \chi_2, \cdots, \chi_{k+l}) \in \mathbb{Z}_{q_1} \times \mathbb{Z}_{q_2} \times \cdots \times \mathbb{Z}_{q_k} \times \mathbb{Z}_{b_1} \times \mathbb{Z}_{b_2} \times \cdots \times \mathbb{Z}_{b_l}$, which represents the value $\chi = \left(\sum_{i=1}^{k} \left| \chi_i \cdot \left(\frac{qb}{q_i} \right)^{-1} \right|_{q_i} \cdot \frac{qb}{q_i} \right) + \left(\sum_{j=k+1}^{k+l} \left| \chi_j \cdot \left(\frac{qb}{b_j} \right)^{-1} \right|_{b_j} \cdot \frac{qb}{b_j} \right) \bmod qb.$

Given $\chi \in \mathbb{Z}_{qb}$, $\mathsf{ModSwitch}_{\mathsf{RNS}}$ from $qb \to q$ is an operation of updating $\chi \in \mathbb{Z}_{qb}$ to some $y \in \mathbb{Z}_q$ where $y \approx \left\lceil \frac{\chi}{b} \right\rfloor$. Unlike in regular modulus switch where we can directly arithmetically divide χ by b and round it, an RNS vector is incompatible with direct arithmetic division on the residues. Therefore, our alternative strategy is to find some small value $\hat{\chi}$ such that $\chi \equiv \hat{\chi} \bmod b$. Once we find such $\hat{\chi}$, then $\chi - \hat{\chi} \bmod qb$ becomes divisible by b (since their difference is some multiple of b), and thus we can compute $\frac{\chi - \hat{\chi}}{b} \approx \left\lceil \frac{\chi}{b} \right\rfloor$. Note that in this computation, the additionally introduced error of modulus switch caused by replacing χ with $\chi - \hat{\chi}$ is equivalent to: $\left| \left\lceil \frac{\chi}{b} \right\rfloor - \frac{\chi - \hat{\chi}}{b} \right| \approx \left\lceil \frac{\hat{\chi}}{b} \right\rfloor$. After the (exact) division of $\frac{\chi - \hat{\chi}}{b}$, we directly replace the modulus qb with q. This direct replacement of modulus is arithmetically allowed because the computation result of $\frac{\chi - \hat{\chi}}{b}$ is guaranteed to be within $-\frac{q}{2}$ and $\frac{q}{2} - 1$ (since $-\frac{qb}{2} \leq \chi \leq \frac{qb}{2} - 1$). Therefore, we can derive the following formula:

$$\frac{\chi - \hat{\chi}}{b} \bmod q = |b^{-1}|_q \cdot (\chi - \hat{\chi}) \bmod q$$

In the above relation, we can arithmetically replace b with $|b^{-1}|_q$, because $\chi - \hat{\chi}$ is divisible by b and b is guaranteed to have an inverse modulo q (since b and q are co-prime). Next, we can compute $|b^{-1}|_q \cdot (\chi - \hat{\chi}) \bmod q$ based on their RNS residues as follows:

$$\begin{aligned}
&|b^{-1}|_q \cdot (\chi - \hat{\chi}) \bmod q \\
&= \left(|b^{-1}|_{q_1} \cdot (\chi_1 - \hat{\chi}_1),\ |b^{-1}|_{q_2} \cdot (\chi_2 - \hat{\chi}_2),\ \cdots,\ |b^{-1}|_{q_k} \cdot (\chi_k - \hat{\chi}_k)\right) \in \mathbb{Z}_{q_1} \times \mathbb{Z}_{q_2} \times \cdots \times \mathbb{Z}_{q_k} \\
&= (y_1, y_2, \cdots, y_k) \ \# \text{ where each } y_i = |b^{-1}|_{q_i} \cdot (\chi_i - \hat{\chi}_i) \bmod q_i
\end{aligned}$$

Now, our task is to derive an expression for some small $\hat{\chi}$ such that $\chi - \hat{\chi}$ is divisible by b. We propose that $\hat{\chi} = |\chi|_b + ub$ for some small integer $|u| \leq \frac{l}{2} + 1$. Then, notice that $\chi - \hat{\chi}$ is divisible by b as follows:

$$|\chi - \hat{\chi}|_b = \Big| |\chi|_b - (|\chi|_b + ub) \Big|_b = |-ub|_b = 0$$

Now, we will derive the RNS vector of $\hat{\chi} \bmod q = |\chi|_b + ub \bmod q$, which is to be plugged into $|b^{-1}|_q \cdot (\chi - \hat{\chi}) \bmod q$. First, we derive the RNS vector of $|\chi|_b$ as follows:

$$(|\chi|_{b_1}, |\chi|_{b_2}, \cdots, |\chi|_{b_l}) = (\chi_{k+1}, \chi_{k+2}, \cdots, \chi_{k+l}) \in \mathbb{Z}_{b_1} \times \mathbb{Z}_{b_2} \times \cdots \times \mathbb{Z}_{b_l}$$

Next, we can compute its fast base conversion from $b \to q$ as follows:

$$\begin{aligned}
&\mathsf{FastBConv}(\{\chi_{k+i}\}_{i=1}^{l}, b, q) \\
&= (\hat{\chi}_1, \hat{\chi}_2, \cdots, \hat{\chi}_k) \in \mathbb{Z}_{q_1} \times \mathbb{Z}_{q_2} \times \cdots \times \mathbb{Z}_{q_k}
\end{aligned}$$

Now, notice that the above RNS residue vector $(\hat{\chi}_1, \hat{\chi}_2, \cdots, \hat{\chi}_k)$ represents the value $\hat{\chi} = |\chi|_b + u \cdot b \bmod q$ (where integer $|u| \leq \frac{l}{2} + 1$), which is our desired formula for $\hat{\chi}$. Therefore, $\hat{\chi} = \mathsf{FastBConv}(\{\chi_{k+i}\}_{i=1}^{l}, b, q)$.

Note that $\hat{\chi} \ll \frac{q}{2} - 1$ and $-\frac{q}{2} \ll \hat{\chi}$, because $\left| |\chi|_b + u \cdot b \right| < \left(\frac{l}{2} + 1\right) \cdot b + \frac{b}{2} \ll \frac{q}{2}$ (here we assume that $b \ll q$, as we assume the modulus switch operation is used to remove only a single prime factor from the large base q). Therefore, the magnitude of the error generated by computing $b^{-1} \cdot (\chi - \hat{\chi})$

is approximately $\left\lceil \frac{\hat{\chi}}{b} \right\rfloor < \left\lceil \frac{\left(\frac{l}{2}+1\right)\cdot b + \frac{b}{2}}{b} \right\rfloor = \left\lceil \frac{lb+3b}{2b} \right\rfloor = \left\lceil \frac{l+3}{2} \right\rfloor < \frac{l}{2}+2.$

We summarize the $\mathsf{ModSwitch}_{\mathsf{RNS}}$ operation as follows:

⟨Summary ??⟩ $\mathsf{ModSwitch}_{\mathsf{RNS}}$

Input: $(\chi_1, \chi_2, \cdots, \chi_{k+l}) \in \mathbb{Z}_{q_1} \times \mathbb{Z}_{q_2} \times \cdots \times \mathbb{Z}_{q_k} \times \mathbb{Z}_{b_1} \times \mathbb{Z}_{b_2} \times \cdots \times \mathbb{Z}_{b_l}$

$\#$ which represents $\chi = \left(\sum_{i=1}^{k} \left| \chi_i \cdot \left(\frac{qb}{q_i}\right)^{-1} \right|_{q_i} \cdot \frac{qb}{q_i} \right) + \left(\sum_{j=k+1}^{k+l} \left| \chi_j \cdot \left(\frac{qb}{b_j}\right)^{-1} \right|_{b_j} \cdot \frac{qb}{b_j} \right) \bmod qb$

Notations

- The RNS vector $(\chi_1, \chi_2, \cdots, \chi_k) \in \mathbb{Z}_{q_1} \times \mathbb{Z}_{q_2} \times \cdots \times \mathbb{Z}_{q_k}$ represents the value: $|\chi|_q \in \mathbb{Z}_q$
- $\mathsf{FastBConv}(\{\chi_{k+1}\}_{i=1}^{l}, b, q) = (\hat{\chi}_1, \hat{\chi}_2, \cdots, \hat{\chi}_k) \in \mathbb{Z}_{q_1} \times \mathbb{Z}_{q_2} \times \cdots \times \mathbb{Z}_{q_k}$
  , which represents the value $\hat{\chi} = |\chi|_b + ub \in \mathbb{Z}_q$ $\#$ where $|u| \leq \frac{l}{2}+1$

Main Steps

$\mathsf{ModSwitch}_{\mathsf{RNS}}(\{\chi_i\}_{i=1}^{k+l}, qb, q)$
$= \{|b^{-1}|_{q_i} \cdot (\chi_i - \hat{\chi}_i) \bmod q_i\}_{i=1}^{k} \in \mathbb{Z}_{q_1} \times \mathbb{Z}_{q_2} \times \cdots \times \mathbb{Z}_{q_k}$

, whose RNS residue vector represents the value $|b^{-1}|_q \cdot (\chi - \hat{\chi}) \bmod q$. The magnitude of noise generated by $\mathsf{ModSwitch}_{\mathsf{RNS}}$ is roughly $\left\lceil \frac{\hat{\chi}}{b} \right\rfloor < \frac{l}{2}+2$.

D-5.4.1 Comparing $\mathsf{ModSwitch}_{\mathsf{RNS}}$, $\mathsf{ModRaise}_{\mathsf{RNS}}$, and $\mathsf{ModDrop}_{\mathsf{RNS}}$

Given a big value $x \in \mathbb{Z}_q$ in an RNS vector, $\mathsf{ModSwitch}_{\mathsf{RNS}}$ reduces its modulus from $q \to q'$ as well as explicitly decreases the modulo value x by the proportion of $\frac{q'}{q}$ (i.e., updates x to $\left\lceil x \cdot \frac{q'}{q} \right\rfloor$). On the other hand, $\mathsf{ModDrop}_{\mathsf{RNS}}$ from $q \to q'$ updates the modulo value from $x \to |x|_{q'}$ (where q' divides q), which is different from decreasing x by the proportion of $\frac{q'}{q}$ like modulus switch. $\mathsf{ModRaise}_{\mathsf{RNS}}$ from $q \to qb$ (where q divides qb) increases the modulus without explicitly modifying the modulo value x, but generates some q-overflow noise. $\mathsf{ModSwitch}_{\mathsf{RNS}}$ and $\mathsf{ModRaise}_{\mathsf{RNS}}$ generate some noise, whereas $\mathsf{ModDrop}_{\mathsf{RNS}}$ does not generate any noise.

D-5.5 RNS-based Decryption

This subsection will explain how to efficiently decrypt RNS-based ciphertexts for BFV, CKKS, and BGV.

D-5.5.1 BFV Decryption: $\mathsf{Dec}^{\mathsf{BFV}}_{\mathsf{RNS}}$

Suppose we have a BFV ciphertext (A, B) such that $B = A \cdot S + \Delta M + E$ (where $\Delta = \left\lfloor \frac{q}{t} \right\rfloor$). We decrypt the ciphertext as follows: $M = \left\lceil \frac{B - A \cdot S}{\Delta} \right\rfloor$ (Summary **??** in **??**). However, RNS does not allow direct division and rounding. Therefore, we need to express this divide-and-round operation in terms of addition and multiplication.

Let's denote $\mathsf{ct}(s) = \Delta m + e + kq$ (i.e., a decryption of ciphertext ct without modulo-q reduction). In this description, we will consider only a single set of coefficients m, e, and k extracted from polynomials M, E, and K for simplicity.

As explained in **??**, modulo arithmetic does not support direct division. Meanwhile, the special relation $\frac{a}{b} \bmod p = a \cdot b^{-1} \bmod p$ holds if b divides a and an inverse of b modulo p exists (i.e., b and p are co-prime). Inspired by this, we can express the decrypted plaintext m as follows:

$$
\begin{aligned}
m &= \left\lceil \frac{|\mathsf{ct}(s)|_q}{\Delta} \right\rfloor = \left\lfloor \frac{|\mathsf{ct}(s)|_q}{\Delta} \right\rfloor + e_r \ \# \text{ where } e_r \in [0, 1] \text{ is a rounding error} \\
&= \left\lfloor |\mathsf{ct}(s)|_q \cdot \frac{t}{q} \right\rfloor + e_r + e_d \ \# \text{ where } e_d = \left\lfloor \frac{|\mathsf{ct}(s)|_q}{\Delta} \right\rfloor - \left\lfloor |\mathsf{ct}(s)|_q \cdot \frac{t}{q} \right\rfloor \text{ is a scaling error} \\
&= \left\lfloor \frac{t \cdot |\mathsf{ct}(s)|_q}{q} \right\rfloor + e_r + e_d \\
&= \frac{t \cdot |\mathsf{ct}(s)|_q - |t \cdot \mathsf{ct}(s)|_q}{q} + e_r + e_d \ \# \text{ where } |t \cdot \mathsf{ct}(s)|_q \equiv t \cdot |\mathsf{ct}(s)|_q \bmod q, \text{ and therefore } t \cdot |\mathsf{ct}(s)|_q - \\
&|t \cdot \mathsf{ct}(s)|_q \text{ is divisible by } q
\end{aligned}
$$

Now, we choose some prime number γ which is co-prime to t and q. Then, we derive the expression for $\gamma \cdot m$ as follows:

$$
\begin{aligned}
\gamma \cdot m &= \gamma \cdot \left\lceil \frac{|\mathsf{ct}(s)|_q}{\Delta} \right\rfloor \\
&= \left\lceil \frac{\gamma \cdot |\mathsf{ct}(s)|_q}{\Delta} \right\rfloor + e'_s \ \# \text{ where } e'_s = \gamma \cdot \left\lceil \frac{|\mathsf{ct}(s)|_q}{\Delta} \right\rfloor - \left\lceil \frac{\gamma \cdot |\mathsf{ct}(s)|_q}{\Delta} \right\rfloor \text{ is a multiplication error} \\
&= \left\lfloor \frac{\gamma \cdot |\mathsf{ct}(s)|_q}{\Delta} \right\rfloor + e'_s + e'_r \ \# \text{ where } e'_r \in [0, 1] \text{ is a rounding error} \\
&= \left\lfloor \gamma \cdot |\mathsf{ct}(s)|_q \cdot \frac{t}{q} \right\rfloor + e'_s + e'_r + e'_d \ \# \text{ where } e'_d = \left(\left\lfloor \frac{\gamma \cdot |\mathsf{ct}(s)|_q}{\Delta} \right\rfloor - \left\lfloor \gamma \cdot |\mathsf{ct}(s)|_q \cdot \frac{t}{q} \right\rfloor \right) \text{ is a scaling} \\
&\text{error} \\
&= \left\lfloor \frac{\gamma \cdot t \cdot |\mathsf{ct}(s)|_q}{q} \right\rfloor + e'_s + e'_r + e'_d \\
&= \frac{\gamma \cdot t \cdot |\mathsf{ct}(s)|_q - |\gamma \cdot t \cdot \mathsf{ct}(s)|_q}{q} + e'_s + e'_r + e'_d
\end{aligned}
$$

Next, we derive the expression for $|\gamma \cdot m|_{\gamma t}$ as follows:

$$
\begin{aligned}
|\gamma \cdot m|_{\gamma t} &= \left| \frac{\gamma \cdot t \cdot |\mathsf{ct}(s)|_q - |\gamma \cdot t \cdot \mathsf{ct}(s)|_q}{q} + e'_s + e'_r + e'_d \right|_{\gamma t} \\
&= \left| \frac{\gamma \cdot t \cdot |\mathsf{ct}(s)|_q - |\gamma \cdot t \cdot \mathsf{ct}(s)|_q}{q} \right|_{\gamma t} + |e'_s|_{\gamma t} + |e'_r|_{\gamma t} + |e'_d|_{\gamma t} \\
&= \left| \frac{\gamma \cdot t \cdot |\mathsf{ct}(s)|_q - |\gamma \cdot t \cdot \mathsf{ct}(s)|_q}{q} \right|_{\gamma t} + e'_s + e'_r + e'_d \ \# \text{ assuming } |e'_s| \ll \frac{\gamma t}{2} \text{ and } |e'_r| \ll \frac{\gamma t}{2} \text{ and}
\end{aligned}
$$

$|e'_d| \ll \frac{\gamma t}{2}$

$= \left|(\gamma \cdot t \cdot |\mathsf{ct}(s)|_q - |\gamma \cdot t \cdot \mathsf{ct}(s)|_q) \cdot q^{-1}\right|_{\gamma t} + e'_s + e'_r + e'_d$ # since $\gamma \cdot t \cdot |\mathsf{ct}(s)|_q - |\gamma \cdot t \cdot \mathsf{ct}(s)|_q$ is divisible by q, and q is co-prime to γt

$= |-|\gamma \cdot t \cdot \mathsf{ct}(s)|_q \cdot q^{-1}|_{\gamma t} + e'_s + e'_r + e'_d$ # since $\gamma \cdot t \cdot |\mathsf{ct}(s)|_q$ is a multiple of γt

$= \left||\gamma \cdot t \cdot \mathsf{ct}(s)|_q\right|_{\gamma t} \cdot |-q^{-1}|_{\gamma t} + e'_s + e'_r + e'_d$

Given the above relation, notice that the computation result of $\mathsf{FastBConv}(\gamma \cdot t \cdot \mathsf{ct}(s), q, \gamma t) \cdot |-q^{-1}|_{\gamma t}$ can be expressed as follows:

$\mathsf{FastBConv}(\gamma \cdot t \cdot \mathsf{ct}(s), q, \gamma t) \cdot |-q^{-1}|_{\gamma t}$

$= \left||\gamma t \cdot \mathsf{ct}(s)|_q + uq\right|_{\gamma t} \cdot |-q^{-1}|_{\gamma t}$ # where $|u| \leq \frac{k}{2} + 1$ for the base moduli $q_1, q_2, \cdots, q_k$

$= \left||\gamma t \cdot \mathsf{ct}(s)|_q \cdot |-q^{-1}|_{\gamma t} - u\right|_{\gamma t}$

$= |\gamma \cdot m|_{\gamma t} - e'_s - e'_r - e'_d - u$ # as we previously showed that $|\gamma \cdot m|_{\gamma t} = \left||\gamma \cdot t \cdot \mathsf{ct}(s)|_q\right|_{\gamma t} \cdot |-q^{-1}|_{\gamma t} + +e'_s + e'_r + e'_d$

$= y$ # let's denote the above expression as y

Then, if e'_s, e'_r, e'_d, u are small enough such that $|e'_s + e'_r + e'_d + u| < \gamma$, then $|y|_\gamma = -e'_s - e'_r - e'_d - u$ (as $|\gamma \cdot m|_{\gamma t} \bmod \gamma = 0$ as a multiple of γ). Therefore, we can effectively remove the noise terms e'_s, e'_r, e'_d, u and derive m as follows:

$\left|(y - |y|_\gamma) \cdot |\gamma^{-1}|_t\right|_t$

$= \left||\gamma \cdot m|_{\gamma t} \cdot |\gamma^{-1}|_t\right|_t$

$= \left||\gamma \cdot m|_t \cdot |\gamma^{-1}|_t\right|_t$ # since $\left||\gamma \cdot m|_{\gamma t}\right|_t = |\gamma \cdot m|_t$

$= |m|_t$

, which is the final decryption of ct we wanted to compute. Let's denote the RNS vector of y as a $(y_\gamma, y_t) \in \mathbb{Z}_\gamma \times \mathbb{Z}_t$. Then, we can efficiently compute the term $\left|(y - |y|_\gamma) \cdot |\gamma^{-1}|_t\right|_t$ as follows:

$\left|(y - |y|_\gamma) \cdot |\gamma^{-1}|_t\right|_t$

$= \left|\Big(\overbrace{\left|y_\gamma \cdot t \cdot |t^{-1}|_\gamma + y_t \cdot \gamma \cdot |\gamma^{-1}|_t\right|_{\gamma t}}^{y} - \overbrace{\left|y_\gamma \cdot t \cdot |t^{-1}|_\gamma + y_t \cdot \gamma \cdot |\gamma^{-1}|_t\right|_\gamma}^{|y|_\gamma}\Big) \cdot |\gamma^{-1}|_t\right|_t$

$= \left|\Big(\overbrace{\left|y_\gamma \cdot t \cdot |t^{-1}|_\gamma + y_t \cdot \gamma \cdot |\gamma^{-1}|_t\right|_t}^{y} - \overbrace{\left|y_\gamma \cdot t \cdot |t^{-1}|_\gamma + y_t \cdot \gamma \cdot |\gamma^{-1}|_t\right|_\gamma}^{|y|_\gamma}\Big) \cdot |\gamma^{-1}|_t\right|_t$ # since $\left||y|_{\gamma t}\right|_t = |y|_t$

$= \left|\Big(\overbrace{\left|y_\gamma \cdot t \cdot |t^{-1}|_\gamma + y_t \cdot \gamma \cdot |\gamma^{-1}|_t\right|_t}^{y} - \overbrace{\left|y_\gamma \cdot t \cdot |t^{-1}|_\gamma\right|_\gamma}^{|y|_\gamma}\Big) \cdot |\gamma^{-1}|_t\right|_t$ # since $y_t \cdot \gamma \cdot |\gamma^{-1}|_t \bmod \gamma = 0$

$$= \left| \Big(\overbrace{\left| y_t \cdot \gamma \cdot |\gamma^{-1}|_t \right|_t}^{y} - \overbrace{\left| y_\gamma \cdot t \cdot |t^{-1}|_\gamma \right|_\gamma}^{|y|_\gamma} \Big) \cdot |\gamma^{-1}|_t \right|_t \quad \# \text{ since } y_\gamma \cdot t \cdot |t^{-1}|_\gamma \bmod t = 0$$

$$= \left| (y_t - y_\gamma) \cdot |\gamma^{-1}|_t \right|_t \quad \# \text{ since } |\gamma \cdot \gamma^{-1}|_t = 1 \text{ and } |t \cdot t^{-1}|_\gamma = 1$$

We summarize $\mathsf{Dec}^{\mathsf{BFV}}_{\mathsf{RNS}}$ as follows:

⟨Summary ??⟩ $\mathsf{Dec}^{\mathsf{BFV}}_{\mathsf{RNS}}$

Input: $\mathsf{ct}(s) = \Delta m + e + kq$

1. Pick some prime number γ which is co-prime to t and q.
2. Compute $\mathsf{FastBConv}(|\gamma \cdot t \cdot \mathsf{ct}(s)|_q, q, \gamma t) \cdot | - q^{-1}|_{\gamma t}$
   $= (y_\gamma, y_t) \in \mathbb{Z}_\gamma \times \mathbb{Z}_t$
3. Compute $|m|_t = \left| (y_t - y_\gamma) \cdot |\gamma^{-1}|_t \right|_t$

D-5.5.2 CKKS and BGV Decryption

CKKS and BGV ciphertexts can be decrypted efficiently by performing the $\mathsf{ModDrop}$ operation (Summary **??** in **??**) to the lowest multiplicative level. After this, there remains only a single ciphertext modulus in the RNS base, so the regular decryption algorithm can be executed efficiently without any RNS components.

D-5.6 BGV's RNS-based Modulus Switch: $\mathsf{ModSwitch}^{\mathsf{BGV}}_{\mathsf{RNS}}$

BGV's RNS-based modulus switch is not computed by $\mathsf{ModSwitch}_{\mathsf{RNS}}$, because BGV's original non-RNS modulus switch (Summary **??** in **??**) is performed in a different manner than BFV or CKKS's non-RNS modulus switch (Summary **??** in **??**). BGV's non-RNS modulus switch is computed as follows:

$$(A', B') = \left(\left\lceil \frac{\hat{q}}{q_l} A \right\rfloor, \left\lceil \frac{\hat{q}}{q_l} \cdot B \right\rfloor \right) \in \mathcal{R}^2_{\langle n, \hat{q} \rangle}$$

$$\epsilon'_A = \hat{q} \cdot A - q_l \cdot A' \quad \# \text{ where } \epsilon'_A \in \mathbb{Z}_{q_l}$$
$$\epsilon'_B = \hat{q} \cdot B - q_l \cdot B' \quad \# \text{ where } \epsilon'_B \in \mathbb{Z}_{q_l}$$

$$H_A = q_l^{-1} \cdot \epsilon'_A \bmod t$$
$$H_B = q_l^{-1} \cdot \epsilon'_B \bmod t$$

$$\hat{\mathsf{ct}} = (\hat{A}, \hat{B}) = (A' + H_A, B' + H_B) \bmod \hat{q}$$

Therefore, BGV's RNS-based modulus switch only needs to compute the above formulas for $\hat{A}$ and $\hat{B}$ based on RNS's $(+, \cdot)$ arithmetic. In the above computations, the only part that cannot be directly computed by RNS-based $(+, \cdot)$ operations is the rounding in $\left\lceil \frac{\hat{q}}{q_l} A \right\rfloor$ and $\left\lceil \frac{\hat{q}}{q_l} \cdot B \right\rfloor$. This rounding can be performed in RNS by using $\mathsf{Dec}^{\mathsf{BFV}}_{\mathsf{RNS}}$, by setting $q = q_l$ and $t = \hat{q}$.

D-5.7 Small Montgomery Reduction Algorithm: SmallMont

One problem of the FastBConv (i.e., the fast base conversion) operation is that it creates a non-negligible noise. Specifically, suppose we use FastBConv to convert the base of $x \in \mathbb{Z}_q$ (where $q = q_1 \cdot q_2 \cdot \cdots \cdot q_k$ moduli) into $c = x + uq \bmod b$ (where $b = b_1 \cdot b_2 \cdot \cdots \cdot b_l$ moduli), where integer $|u| \leq \frac{k}{2} + 1$. Then, the noise generated by this conversion is between $-\left(\frac{k}{2}+1\right) \cdot q \bmod b$ and $\left(\frac{k}{2}+1\right) \cdot q \bmod b$. To reduce this noise, we will explain the small Montgomery algorithm (SmallMont) which reduces the noise generated by fast base conversion from uq to $u'q$, such that $u' \in \{-1, 0, 1\}$. The small Montgomery algorithm is designed as follows:

⟨Summary ??⟩ Fast Modulo Reduction: SmallMont

Input: $c = (c_1, c_2, \cdots, c_l, c_{l+1}) \in \mathbb{Z}_{b_1} \times \mathbb{Z}_{b_2} \times \cdots \times \mathbb{Z}_{b_l} \times \mathbb{Z}_{b_\alpha}$ # b_α is a prime and co-prime to b, where $b = \prod_{i=1}^{l} b_i$

, where $c = \mathsf{FastBConv}(|b_\alpha \cdot x|_q, q, bb_\alpha) = |b_\alpha \cdot x|_q + uq$ # where $x \in \mathbb{Z}_q$ and integer $|u| \leq \frac{k}{2}+1$

Main Steps

$\mathsf{SmallMont}(c, bb_\alpha, b_\alpha, q)$:

1. $c' = |c \cdot q^{-1}|_{b_\alpha}$
2. For each $i \in [1, l]$, compute $r_i = \left|(c_{b_i} - |q|_{b_i} \cdot c') \cdot b_\alpha^{-1}\right|_{b_i}$

Output: $r = (\overbrace{r_1, r_2, \cdots, r_l}^{l}) \in \overbrace{\mathbb{Z}_{b_1} \times \mathbb{Z}_{b_2} \times \cdots \times \mathbb{Z}_{b_l}}^{l}$ # without $r_\alpha \in \mathbb{Z}_{b_\alpha}$

The output satisfies the relation: $r = x + u'q \bmod b$ (where $u' \in \{-1, 0, 1\}$)

Proof.

1. Given $c' = |c \cdot q^{-1}|_{b_\alpha}$, notice that $c - q \cdot c'$ is exactly divisible by b_α as shown below:
   $c - q \cdot c' \bmod b_\alpha$
   $= c - q \cdot |c \cdot q^{-1}|_{b_\alpha} \bmod b_\alpha$ # substituting $c' = |c \cdot q^{-1}|_{b_\alpha}$
   $= c - c \bmod b_\alpha$ # by canceling out $|q|_{b_\alpha}$ and $|q|_{b_\alpha}^{-1}$
   $= 0 \bmod b_\alpha$

   Since $c - q \cdot c' = 0 \bmod b_\alpha$, this implies that $c - q \cdot c'$ is a multiple of b_α (i.e., $c - q \cdot c'$ is exactly divisible by b_α). This also implies that $\frac{c - q \cdot c'}{b_\alpha}$ is an integer.

2. Given $c = |b_\alpha \cdot x|_q + uq \bmod b$ and $c' = |c \cdot q^{-1}|_{b_\alpha}$, we can express $\frac{c - q \cdot c'}{b_\alpha} \bmod b$ as follows:
   $\left|\frac{c - q \cdot c'}{b_\alpha}\right|_b$
   $= \left|\frac{c - q \cdot |c \cdot q^{-1}|_{b_\alpha}}{b_\alpha}\right|_b$ # by substituting $c' = |c \cdot q^{-1}|_{b_\alpha}$

$$= \left| \frac{|b_\alpha \cdot x|_q + uq - q \cdot \left| |b_\alpha \cdot x|_q + uq \right|_b \cdot q^{-1} \Big|_{b_\alpha}}{b_\alpha} \right|_b \quad \# \text{ by substituting } c = \left| |b_\alpha \cdot x|_q + uq \right|_b$$

$$= \left| \frac{b_\alpha \cdot x + vq + uq - q \cdot \left| |b_\alpha \cdot x + vq + uq|_b \cdot q^{-1} \right|_{b_\alpha}}{b_\alpha} \right|_b \quad \# \text{ by rewriting } |b_\alpha \cdot x|_q \text{ as } b_\alpha \cdot x + vq \text{ (where } v \text{ is some integer representing the } q\text{-overflows of } b_\alpha \cdot x)$$

$$= \left| x + \frac{vq + uq - q \cdot \left| |b_\alpha \cdot x + vq + uq|_b \cdot q^{-1} \right|_{b_\alpha}}{b_\alpha} \right|_b \quad \# \text{ since } x = \frac{b_\alpha \cdot x}{b_\alpha}$$

$$= \left| x + q \cdot \frac{v + u - \left| |b_\alpha \cdot x + vq + uq|_b \cdot q^{-1} \right|_{b_\alpha}}{b_\alpha} \right|_b \quad \# \text{ taking out the common multiple } q$$

The above computation result is guaranteed to be an integer (as we proved in the proof step 1). And q and b_α are co-prime (by the input definition). This leads to the conclusion that $\frac{v + u - \left| |b_\alpha \cdot x + vq + uq|_b \cdot q^{-1} \right|_{b_\alpha}}{b_\alpha}$ is guaranteed to be an integer. Therefore, if we choose b_α (i.e., a prime and co-prime to both q and b) as a sufficiently large value, then $\frac{v + u - \left| |b_\alpha \cdot x + vq + uq|_b \cdot q^{-1} \right|_{b_\alpha}}{b_\alpha}$ will converge to $\{-1, 0, 1\}$. This is because as b_α increases: (1) v grows slower than b_α (since $|b_\alpha \cdot x|_q = b_\alpha \cdot x + vq$); (2) the magnitude of u stays smaller than $\frac{k}{2} + 1$ (as integer $|u| \leq \frac{k}{2} + 1$); and (3) $\left| |b_\alpha \cdot x + vq + uq|_b \cdot q^{-1} \right|_{b_\alpha}$ is guaranteed to be an integer between $\left[-\frac{b_\alpha + 1}{2}, \frac{b_\alpha}{2} - 1 \right]$. In conclusion, if b_α is sufficiently large, then we get the following relation:

$\left| \frac{c - q \cdot c'}{b_\alpha} \right|_b = x + u'q \bmod b$ # where $u' \in \{-1, 0, 1\}$

Also, the following is true:

$\frac{c - q \cdot c'}{b_\alpha} \bmod b = (c - q \cdot c') \cdot b_\alpha^{-1} \bmod b$ # because b_α divides $c - q \cdot c'$ and b_α is co-prime to b

3. It is possible to express the final output $x + u'q \bmod b$ as an RNS vector with the residues of the base moduli $(b_1, \cdots, b_l)$. For this, we convert $(c - q \cdot c') \cdot b_\alpha^{-1}$ into the RNS vector $(r_1, r_2, \cdots, r_l) \in \mathbb{Z}_{b_1} \times \mathbb{Z}_{b_2} \times \cdots \times \mathbb{Z}_{b_l}$ by computing the following for each $i \in [1, l]$:
$r_i = |(c - q \cdot c') \cdot b_\alpha^{-1}|_{b_i}$
$= |(c_{b_i} - |q|_{b_i} \cdot c') \cdot b_\alpha^{-1}|_{b_i}$

□

D-5.7.1 Improving FastBConv by Using SmallMont

Notice that by using SmallMont in Summary **??**, the accuracy of the raw output of FastBConv$(x, q, b) = |x + uq|_b$ (where integer $|u| \leq \frac{k}{2} + 1$) is improved to $|x + u'q|_b$ (where $u' \in \{-1, 0, 1\}$) as follows:

$\mathsf{SmallMont}(\mathsf{FastBConv}(|b_\alpha \cdot x|_q, q, bb_\alpha), bb_\alpha, b_\alpha, q)$
$\mathsf{SmallMont}(\left||b_\alpha \cdot x|_q + uq\right|_{bb_\alpha}, bb_\alpha, b_\alpha, q)$
$= |x + u'q|_b$ # where $u' \in \{-1, 0, 1\}$

D-5.8 Exact Fast Base Conversion: FastBConvEx

FastBConv (??) converts an input value x's base moduli from $q \to b$, but generates a noise equivalent to uq mod b where integer $|u| \leq \frac{k}{2} + 1$. If we use FastBConv with SmallMont (??), we can reduce the generated noise from uq to $u'q$ where $u' \in \{-1, 0, 1\}$. In this subsection, we introduce FastBConvEx, an algorithm for an exact fast base conversion that can eliminate the entire noise. However, using FastBConvEx has a restriction that the input value x should be relatively much smaller than its modulus. This is different from the case of using FastBConv with SmallMont which has no restriction on the input x (i.e., x can be any value within its modulus range). FastBConvExis designed as follows:

⟨Summary ??⟩ Fast Exact Base Conversion: FastBConvEx

Input: $x = (x_1, x_2, \cdots, x_l, x_\alpha) \in \mathbb{Z}_{b_1} \times \mathbb{Z}_{b_2} \times \cdots \times \mathbb{Z}_{b_l} \times \mathbb{Z}_{b_\alpha}$

Requirement: The size of b_α should be $b_\alpha \geq 2 \cdot (l + \lambda)$, where $|x|_b = x + \mu \cdot b$, and $\mu \in [-\lambda, \lambda]$ (i.e., λ and $-\lambda$ are the maximum and minimum possible values of μ).
The constraint that $b_\alpha > \mu$ implies that the input x should be much smaller than its modulus bb_α (i.e., $|x| \ll \frac{bb_\alpha}{2}$)

Main Steps
1. $\hat{x} = |x|_b = \mathsf{ModDrop}(x, bb_\alpha, b)$
2. $x_\alpha = |x|_{b_\alpha} = \mathsf{ModDrop}(x, bb_\alpha, b_\alpha)$
3. $\gamma = |(\mathsf{FastBConv}(\hat{x}, b, b_\alpha) - x_\alpha) \cdot b^{-1}|_{b_\alpha}$
4. $\mathsf{FastBConvEx}(x, bb_\alpha, q) = \left|\mathsf{FastBConv}(\hat{x}, b, q) - \gamma \cdot b\right|_q = |x|_q$

We will prove why $\left|\mathsf{FastBConv}(\hat{x}, b, q) - \gamma \cdot b\right|_q = |x|_q$.

Proof.
1. $\mathsf{FastBConv}(\hat{x}, b, b_\alpha)$
   $= |\hat{x} + ub|_{b_\alpha}$ # where integer $|u| \leq \frac{l}{2} + 1$
   $= \left||x|_b + ub\right|_{b_\alpha}$ # since $|x|_b = \hat{x}$ by definition
   $= |x + \mu b + ub|_{b_\alpha}$ # since $|x|_b = x + \mu b$ by definition

2. $\gamma = |(\mathsf{FastBConv}(\hat{x}, b, b_\alpha) - x_\alpha) \cdot b^{-1}|_{b_\alpha}$
   $= |(\mathsf{FastBConv}(\hat{x}, b, b_\alpha) - x_\alpha - \mu b) \cdot b^{-1} + \mu|_{b_\alpha}$ # by adding $|(-\mu b + \mu b) \cdot b^{-1}|_{b_\alpha}$
   $= |(x + \mu b + ub - x_\alpha - \mu b) \cdot b^{-1} + \mu|_{b_\alpha}$ # step 1 proved $\mathsf{FastBConv}(\hat{x}, b, b_\alpha) = |x + \mu b + u \cdot b|_{b_\alpha}$
   $= |u + \mu|_{b_\alpha}$

$= u+\mu$ # because $-\frac{b_\alpha}{2} \le u+\mu \le \frac{b_\alpha}{2}-1$ (since $u+\mu < l+\lambda \le \frac{b_\alpha}{2}$, and $-\frac{b_\alpha}{2} \le -(l+\lambda) < u+\mu$)

3. $\mathsf{FastBConvEx}(x, b, q) = \left|\mathsf{FastBConv}(\hat{x}, b, q) - \gamma \cdot b)\right|_q$
   $= \left|\hat{x} + ub - \gamma \cdot b)\right|_q$ # applying $\mathsf{FastBConv}(\hat{x}, b, q) = \hat{x} + ub$
   $= \left|(x + \mu b) + ub - \gamma \cdot b)\right|_q$ # applying $\hat{x} = |x|_b = x + \mu b$
   $= \left|(x + \mu b) + ub - (u + \mu) \cdot b)\right|_q$ # applying $\gamma = u + \mu$ from proof step 2
   $= |x + \mu b + ub - ub - \mu b|_q$
   $= |x|_q$

□

Necessity of the Centered (i.e., Signed) Residue Representation: In the proof step 2, we treated $|u+\mu|_{b_\alpha} = u+\mu$. To remove the modulo reduction operation, the canonical (i.e., unsigned) residue representation is inappropriate, because if $u + \mu$ becomes negative, then the residue will underflow and have to be wrapped around, which requires a modulo reduction operation. To prevent the occurrence of both overflow and underflow cases, we need the centered (i.e., signed) residue representation.

D-5.9 Decomposing Multiplication: $\mathsf{DecompMult}_{\mathsf{RNS}}$

In FHE, gadget decomposition (**??**) is used to compute ciphertext-to-plaintext multiplication with a small noise. For example, BFV and CKKS's homomorphic key switching (Summary **??** in **??**) uses gadget decomposition to compute $\mathsf{RLWE}_{S',\sigma}(\Delta M) = B + A \cdot \mathsf{RLWE}_{S',\sigma}(S)$ with a small noise (where each coefficient of the polynomial A can be any value within the range of the ciphertext modulus q). As another example, the relinearization process of ciphertext-to-ciphertext multiplication in BFV (Summary **??** in **??**), CKKS (Summary **??** in **??**), and BGV (Summary **??** in **??**) uses gadget decomposition to derive the synthetic ciphertext $\mathsf{RLWE}_{S',\sigma}(D_2 \cdot S^2)$ when computing $\mathsf{RLWE}_{S',\sigma}(\Delta^2 M^{\langle 1 \rangle} M^{\langle 2 \rangle}) = D_0 + D_1 \cdot S + D_2 \cdot S^2 = \mathsf{ct}_\alpha + \mathsf{ct}_\beta$, where $\mathsf{ct}_\alpha = (D_0, D_1)$, $\mathsf{ct}_\beta = \mathsf{RLWE}_{S',\sigma}(D_2 \cdot S^2)$, $D_0 = B_1 B_2$, $D_1 = A_1 B_2 + A_2 B_1$, and $D_2 = A_1 A_2$. Using gadget decomposition, we showed the following relations:

$\mathsf{RLWE}_{S',\sigma}(A \cdot S) = \langle \mathsf{Decomp}^{\beta,l}(A),\ \mathsf{RLev}^{\beta,l}_{S',\sigma}(S) \rangle$ # used in key switching
$\mathsf{RLWE}_{S',\sigma}(D_2 \cdot S^2) = \langle \mathsf{Decomp}^{\beta,l}(D_2),\ \mathsf{RLev}^{\beta,l}_{S,\sigma}(S^2) \rangle$ # used in relinearization

However, if we convert a value (e.g., x) into an RNS vector, then it cannot be directly expressed in a gadget-decomposed form based on the β and l parameters. Instead, given the relationship between the value x and its RNS residues is $x = \sum_{i=1}^{k} x \cdot \frac{q}{q_i} \cdot \left|\left(\frac{q}{q_i}^{-1}\right)\right|_{q_i} \mod q$, we can treat each RNS residue as a gadget-decomposed element. For example, suppose our goal is to decompose $\mathsf{RLWE}_{S',\sigma}(A \cdot S)$, where the RNS vector of $A = (A_1, A_2, \cdots, A_k)$ and whose base moduli are $(q_1, q_2, \cdots, q_k)$. We can decompose $\mathsf{RLWE}_{S',\sigma}(A \cdot S)$ as follows:

$\mathsf{RLWE}_{S',\sigma}(A \cdot S) \mod q$
$= \mathsf{RLWE}_{S',\sigma}\left(S \cdot \left(A_1 \frac{q}{q_1} \cdot \left|\left(\frac{q}{q_1}\right)^{-1}\right|_{q_1} + A_2 \frac{q}{q_2} \cdot \left|\left(\frac{q}{q_2}\right)^{-1}\right|_{q_2} + \cdots + A_k \frac{q}{q_k} \cdot \left|\left(\frac{q}{q_k}\right)^{-1}\right|_{q_k}\right)\right) \mod q$

$$= \mathsf{RLWE}_{S',\sigma}\left(S \cdot A_1 \frac{q}{q_1} \cdot \left|\left(\frac{q}{q_1}\right)^{-1}\right|_{q_1}\right) + \mathsf{RLWE}_{S',\sigma}\left(S \cdot A_2 \frac{q}{q_2} \cdot \left|\left(\frac{q}{q_2}\right)^{-1}\right|_{q_2}\right) +$$
$$\cdots + \mathsf{RLWE}_{S',\sigma}\left(S \cdot A_k \frac{q}{q_k} \cdot \left|\left(\frac{q}{q_k}\right)^{-1}\right|_{q_k}\right) \bmod q$$

$$= A_1 \cdot \mathsf{RLWE}_{S',\sigma}\left(S \cdot \frac{q}{q_1} \cdot \left|\left(\frac{q}{q_1}\right)^{-1}\right|_{q_1}\right) + A_2 \cdot \mathsf{RLWE}_{S',\sigma}\left(S \cdot \frac{q}{q_2} \cdot \left|\left(\frac{q}{q_2}\right)^{-1}\right|_{q_2}\right) +$$
$$\cdots + A_k \cdot \mathsf{RLWE}_{S',\sigma}\left(S \cdot \frac{q}{q_k} \cdot \left|\left(\frac{q}{q_k}\right)^{-1}\right|_{q_k}\right) \bmod q$$

$$= \sum_{i=1}^{k}\left(A_i \cdot \mathsf{RLWE}_{S',\sigma}\left(S \cdot \frac{q}{q_i} \cdot \left|\left(\frac{q}{q_i}\right)^{-1}\right|_{q_i}\right)\right) \bmod q$$

, where $\left\{\mathsf{RLWE}_{S',\sigma}\left(S \cdot \frac{q}{q_i} \cdot \left|\left(\frac{q}{q_i}\right)^{-1}\right|_{q_i}\right)\right\}_{i=1}^{k}$ can be pre-generated as RNS key-switching keys.

Applying the same reasoning as the above, we can also derive the following for relinearization:
$$\mathsf{RLWE}_{S,\sigma}(D_2 \cdot S^2) = \sum_{i=1}^{k}\left(D_{2,i} \cdot \mathsf{RLWE}_{S,\sigma}\left(S^2 \cdot \frac{q}{q_i} \cdot \left|\left(\frac{q}{q_i}\right)^{-1}\right|_{q_i}\right)\right) \bmod q$$

, where $\left\{\mathsf{RLWE}_{S,\sigma}\left(S^2 \cdot \frac{q}{q_i} \cdot \left|\left(\frac{q}{q_i}\right)^{-1}\right|_{q_i}\right)\right\}_{i=1}^{k}$ can be pre-generated as relinearization keys.

RNS-based multiplication decomposition is summarized as follows:

⟨Summary ??⟩ $\mathsf{DecompMult}_{\mathsf{RNS}}$

For key-switching:

Input: $A = (A_1, A_2, \cdots, A_k) \in \mathcal{R}_{\langle n,q_1\rangle} \times \mathcal{R}_{\langle n,q_2\rangle} \times \cdots \times \mathcal{R}_{\langle n,q_k\rangle}$,
$$S_{\langle S',\mathsf{RNS}\rangle} = \left\{\mathsf{RLWE}_{S',\sigma}\left(S \cdot \frac{q}{q_i} \cdot \left|\left(\frac{q}{q_i}\right)^{-1}\right|_{q_i}\right)\right\}_{i=1}^{k} \quad \text{\# key-switching keys}$$

$\mathsf{DecompMult}_{\mathsf{RNS}}(A, S_{\langle S',\mathsf{RNS}\rangle}) = \mathsf{RLWE}_{S',\sigma}(A \cdot S)$
$$= \sum_{i=1}^{k}\left(A_i \cdot \mathsf{RLWE}_{S',\sigma}\left(S \cdot \frac{q}{q_i} \cdot \left|\left(\frac{q}{q_i}\right)^{-1}\right|_{q_i}\right)\right) \bmod q$$

For relinearization:

Input: $D_2 = (D_{2,1}, D_{2,2}, \cdots, D_{2,k}) \in \mathcal{R}_{\langle n,q_1\rangle} \times \mathcal{R}_{\langle n,q_2\rangle} \times \cdots \times \mathcal{R}_{\langle n,q_k\rangle}$,

$$S^2_{\langle S,\mathsf{RNS}\rangle} = \left\{ \mathsf{RLWE}_{S,\sigma}\left(S^2 \cdot \frac{q}{q_i} \cdot \left| \left(\frac{q}{q_i}\right)^{-1} \right|_{q_i} \right) \right\}_{i=1}^{k} \quad \# \text{ relinearization keys}$$

$$\mathsf{DecompMult}_{\mathsf{RNS}}(D_2, S^2_{\langle S,\mathsf{RNS}\rangle}) = \mathsf{RLWE}_{S,\sigma}(D_2 \cdot S^2)$$

$$= \sum_{i=1}^{k} \left(D_{2,i} \cdot \mathsf{RLWE}_{S,\sigma}\left(S^2 \cdot \frac{q}{q_i} \cdot \left| \left(\frac{q}{q_i}\right)^{-1} \right|_{q_i} \right) \right) \bmod q$$

D-5.10 Applying RNS Techniques to FHE Operations

This subsection will explain how the RNS primitives we have learned so far are used to handle FHE operations for RNS-based ciphertexts in BFV, CKKS, and BGV.

D-5.10.1 Addition and Multiplication of Polynomials

In BFV, CKKS, and BGV, ciphertext-to-plaintext addition, ciphertext-to-ciphertext addition, and ciphertext-to-plaintext multiplication are performed by only involving modulo additions and multiplications among polynomial coefficients. Therefore, we can represent each polynomial coefficient as an RNS residue vector and compute coefficient-wise additions and multiplications by using RNS-based addition and multiplication of residues as explained in Summary **??** (**??**). For example, suppose we have the following two polynomials $P^{\langle 1\rangle}$ and $P^{\langle 2\rangle}$:

$$P^{\langle 1\rangle} = \sum_{a=0}^{n-1} c_a^{\langle 1\rangle} \cdot X^a \in \mathcal{R}_{\langle n,q\rangle}$$
$$P^{\langle 2\rangle} = \sum_{b=0}^{n-1} c_b^{\langle 2\rangle} \cdot X^b \in \mathcal{R}_{\langle n,q\rangle}$$

In the RNS-variant FHE schemes, we express each polynomial's each coefficient as an RNS residue vector as follows:

$$c_a^{\langle 1\rangle} = (c_{a,1}^{\langle 1\rangle}, c_{a,2}^{\langle 1\rangle}, \cdots, c_{a,k}^{\langle 1\rangle}) \in \mathbb{Z}_{q_1} \times \mathbb{Z}_{q_2} \times \cdots \times \mathbb{Z}_{q_k} \quad \# \text{ for } a \in [0, n-1]$$
$$c_b^{\langle 2\rangle} = (c_{b,1}^{\langle 2\rangle}, c_{b,2}^{\langle 2\rangle}, \cdots, c_{b,k}^{\langle 2\rangle}) \in \mathbb{Z}_{q_1} \times \mathbb{Z}_{q_2} \times \cdots \times \mathbb{Z}_{q_k} \quad \# \text{ for } b \in [0, n-1]$$

Given the above RNS setup, when we add or multiply two polynomials, each coefficient-to-coefficient addition is computed as element-wise additions of two RNS residue vectors as follows:

$$c_a^{\langle 1\rangle} + c_b^{\langle 2\rangle} \equiv \sum_{i=1}^{k} (c_{a,i}^{\langle 1\rangle} + c_{b,i}^{\langle 2\rangle}) y_i z_i \bmod q \quad \# \text{ where } y_i = \frac{q}{q_i},\ z_i = |y_i^{-1}|_{q_i}$$
$$\iff (c_{a,1}^{\langle 1\rangle} + c_{b,1}^{\langle 2\rangle},\ c_{a,2}^{\langle 1\rangle} + c_{b,2}^{\langle 2\rangle}, \cdots,\ c_{a,k}^{\langle 1\rangle} + c_{b,k}^{\langle 2\rangle}) \in \mathbb{Z}_{q_1} \times \mathbb{Z}_{q_2} \times \cdots \times \mathbb{Z}_{q_k}$$

Similarly, each coefficient-to-coefficient multiplication is computed as element-wise multiplications of two RNS residue vectors as follows:

$$c_a^{\langle 1\rangle} \cdot c_b^{\langle 2\rangle} \equiv \sum_{i=1}^{k} (c_{a,i}^{\langle 1\rangle} \cdot c_{b,i}^{\langle 2\rangle}) y_i z_i \bmod q$$

$\iff (c^{\langle 1 \rangle}_{a,1} \cdot c^{\langle 2 \rangle}_{b,1},\ c^{\langle 1 \rangle}_{a,2} \cdot c^{\langle 2 \rangle}_{b,2}, \cdots,\ c^{\langle 1 \rangle}_{a,k} \cdot c^{\langle 2 \rangle}_{b,k}) \in \mathbb{Z}_{q_1} \times \mathbb{Z}_{q_2} \times \cdots \times \mathbb{Z}_{q_k}$

Using the above isomorphism, we can efficiently compute ciphertext-to-plaintext addition, ciphertext-to-ciphertext addition, and ciphertext-to-plaintext multiplication of big polynomial coefficients (e.g., 1000 bits big) based on small RNS residues (e.g., 30 bits each).

D-5.10.2 Key Switching

In BFV or CKKS, an RNS-based ciphertext's key-switching operation from $S \rightarrow S'$ is performed by computing the following formula in RNS vectors:
$\mathsf{RLWE}_{S',\sigma}(\Delta M) = B + \langle \mathsf{Decomp}^{\beta,l}(A),\ \mathsf{RLev}^{\beta,l}_{S',\sigma}(S) \rangle$

In the above formula, the computation of $\langle \mathsf{Decomp}^{\beta,l}(A),\ \mathsf{RLev}^{\beta,l}_{S',\sigma}(S) \rangle$ can be performed by using $\mathsf{DecompMult}_{\mathsf{RNS}}$ (Summary **??** in **??**), after which B can be added to it by using regular RNS-based addition.

Similarly, in the case of the BGV, an RNS-based key-switching operation on a ciphertext from $S \rightarrow S'$ is performed by computing the following in RNS vectors:
$\mathsf{RLWE}_{S',\sigma}(M) = B + \langle \mathsf{Decomp}^{\beta,l}(A),\ \mathsf{RLev}^{\beta,l}_{S',\sigma}(S) \rangle$

D-5.10.3 Input Slot Rotation

In BFV or CKKS, an RNS-based ciphertext's input slot rotation is performed by computing the following formulas in RNS vectors:

1. $\mathsf{RLWE}_{S(X^{J(h)}),\sigma}(\Delta M(X^{J(h)})) = (A(X^{J(h)}),\ B(X^{J(h)}))$ # where $J(h) = 5^h \bmod 2n$
2. Key-switch $\mathsf{RLWE}_{S(X^{J(h)}),\sigma}(\Delta M(X^{J(h)}))$ to $\mathsf{RLWE}_{S(X),\sigma}(\Delta M(X^{J(h)}))$

Step 1 is equivalent to re-positioning the coefficients within each polynomial and flipping their signs whenever they cross the boundary of the n-th degree term. This step can be done with RNS-based coefficients by moving around each set of RNS residue vectors as a whole whenever the coefficient they represent is re-positioned to a new degree term, and flipping the signs of the residues in the same RNS vector altogether whenever their representing coefficient's sign is to be flipped. Step 2's RNS-based key switching can be done in the same way as explained in the previous subsection (**??**).

Similarly, in BGV, an RNS-based ciphertext's input slot rotation is performed by computing the following formulas in RNS vectors:

1. $\mathsf{RLWE}_{S(X^{J(h)}),\sigma}(M(X^{J(h)})) = (A(X^{J(h)}),\ B(X^{J(h)}))$ # where $J(h) = 5^h \bmod 2n$
2. Key-switch $\mathsf{RLWE}_{S(X^{J(h)}),\sigma}(M(X^{J(h)}))$ to $\mathsf{RLWE}_{S(X),\sigma}(M(X^{J(h)}))$

We can compute the above formulas in RNS by using the same strategy explained for BFV or CKKS.

D-5.10.4 BFV's Ciphertext-to-Ciphertext Multiplication

BFV's ciphertext-to-ciphertext multiplication (Summary **??** in **??**) comprises $\mathsf{ModRaise} \rightarrow$ polynomial multiplication $\rightarrow$ relinearization $\rightarrow$ rescaling, where the order of relinearization and rescaling can be swapped. In RNS-based ciphertext-to-ciphertext multiplication, we will swap the order of these two steps. The procedure is as follows: (1) $\mathsf{ModRaise}_{\mathsf{RNS}}$ from $q \rightarrow qb$; (2) polynomial multiplication; (3) constant multiplication by t; (4) $\mathsf{ModSwitch}$ from $qb \rightarrow b$; (5) $\mathsf{FastBConvEx}$ from $b \rightarrow q$; and (6) relinearization. Among these, step $3 \sim 5$ corresponds to the rescaling operation. We will explain how each of these steps works.

1. **$\mathsf{ModRaise}_{\mathsf{RNS}}$ from $q \rightarrow qbb_\alpha$:**
   Let b be a new RNS base where $b > \Delta$ so that qb is large enough to prevent a multiplied scaled plaintext in ciphertexts (i.e., $\Delta^2 M^{\langle 1 \rangle} M^{\langle 2 \rangle}$) from exceeding its allowed limit (Summary **??** in **??**) during ciphertext-to-ciphertext multiplication. b_α is also added for exact fast base conversion to be performed later. Specifically, we mod-raise the modulus of each polynomial coefficient of ciphertexts $(A^{\langle 1 \rangle}, B^{\langle 1 \rangle})$ and $(A^{\langle 2 \rangle}, B^{\langle 2 \rangle})$ as follows:
   $(\hat{A}^{\langle 1 \rangle}, \hat{B}^{\langle 1 \rangle}) = (A^{\langle 1 \rangle} + U_A^{\langle 1 \rangle} q, B^{\langle 1 \rangle} + U_B^{\langle 1 \rangle} q) \bmod qbb_\alpha$
   $(\hat{A}^{\langle 2 \rangle}, \hat{B}^{\langle 2 \rangle}) = (A^{\langle 2 \rangle} + U_A^{\langle 2 \rangle} q, B^{\langle 2 \rangle} + U_B^{\langle 2 \rangle} q) \bmod qbb_\alpha$

   , where each coefficient of $U_A^{\langle 1 \rangle}, U_B^{\langle 1 \rangle}, U_A^{\langle 2 \rangle}, U_B^{\langle 2 \rangle}$ is either $\{-1, 0, 1\}$. Decrypting these two (noisy) ciphertexts with the private key S would give the following outputs:
   $\hat{A}^{\langle 1 \rangle} \cdot S + \hat{B}^{\langle 1 \rangle} \bmod qbb_\alpha$
   $= (A^{\langle 1 \rangle} + U_A^{\langle 1 \rangle} q) \cdot S + (B^{\langle 1 \rangle} + U_B^{\langle 1 \rangle} q) \bmod qbb_\alpha$
   $= A^{\langle 1 \rangle} \cdot S + B^{\langle 1 \rangle} + U_A^{\langle 1 \rangle} q \cdot S + U_B^{\langle 1 \rangle} q \bmod qbb_\alpha$
   $= \Delta M^{\langle 1 \rangle} + E^{\langle 1 \rangle} + U_A^{\langle 1 \rangle} q \cdot S + U_B^{\langle 1 \rangle} q + K^{\langle 1 \rangle} q \bmod qbb_\alpha$ # where $+K^{\langle 1 \rangle} q$ is the q-overflows of the decryption process

   $\hat{A}^{\langle 2 \rangle} \cdot S + \hat{B}^{\langle 2 \rangle} \bmod qbb_\alpha$
   $= (A^{\langle 2 \rangle} + U_A^{\langle 2 \rangle} q) \cdot S + (B^{\langle 2 \rangle} + U_B^{\langle 2 \rangle} q) \bmod qbb_\alpha$
   $= A^{\langle 2 \rangle} \cdot S + B^{\langle 2 \rangle} + U_A^{\langle 2 \rangle} q \cdot S + U_B^{\langle 2 \rangle} q \bmod qbb_\alpha$
   $= \Delta M^{\langle 2 \rangle} + E^{\langle 2 \rangle} + U_A^{\langle 2 \rangle} q \cdot S + U_B^{\langle 2 \rangle} q + K^{\langle 2 \rangle} q \bmod qbb_\alpha$

2. **Polynomial Multiplication:**
   Compute $(\hat{B}^{\langle 1 \rangle} \hat{B}^{\langle 2 \rangle}, \hat{A}^{\langle 1 \rangle} \hat{B}^{\langle 2 \rangle} + \hat{A}^{\langle 2 \rangle} B^{\langle 1 \rangle}, \hat{A}^{\langle 1 \rangle} \hat{A}^{\langle 2 \rangle}) \bmod qbb_\alpha$, whose decryption relation is as follows:
   $\hat{B}^{\langle 1 \rangle} \hat{B}^{\langle 2 \rangle} + (\hat{A}^{\langle 1 \rangle} \hat{B}^{\langle 2 \rangle} + \hat{A}^{\langle 2 \rangle} B^{\langle 1 \rangle}) \cdot S + (\hat{A}^{\langle 1 \rangle} \hat{A}^{\langle 2 \rangle}) \cdot S^2 \bmod qbb_\alpha$
   $= (\hat{A}^{\langle 1 \rangle} \cdot S + \hat{B}^{\langle 1 \rangle}) \cdot (\hat{A}^{\langle 2 \rangle} \cdot S + \hat{B}^{\langle 2 \rangle}) \bmod qbb_\alpha$
   $= (\Delta M^{\langle 1 \rangle} + E^{\langle 1 \rangle} + U_A^{\langle 1 \rangle} q \cdot S + U_B^{\langle 1 \rangle} q + K^{\langle 1 \rangle} q) \cdot (\Delta M^{\langle 2 \rangle} + E^{\langle 2 \rangle} + U_A^{\langle 2 \rangle} q \cdot S + U_B^{\langle 2 \rangle} q + K^{\langle 2 \rangle} q) \bmod qbb_\alpha$

3. **Constant Multiplication by t:**
   Step $3 \sim 5$ are equivalent to rescaling the plaintext's scaling factor from $\Delta^2 \rightarrow \Delta$ as well as switching the ciphertext's modulus from qbb_α to q. In this step, we multiply t to each coefficient of the resulting polynomials from the previous step as follows:
   $(t \cdot \hat{B}^{\langle 1 \rangle} \hat{B}^{\langle 2 \rangle},\ t \cdot \hat{A}^{\langle 1 \rangle} \hat{B}^{\langle 2 \rangle} + t \cdot \hat{A}^{\langle 2 \rangle} B^{\langle 1 \rangle},\ t \cdot \hat{A}^{\langle 1 \rangle} \hat{A}^{\langle 2 \rangle}) \bmod qbb_\alpha$

, which is equivalent to a ciphertext encrypting the following plaintext:
$t \cdot (\Delta M^{\langle 1 \rangle} + E^{\langle 1 \rangle} + U_A^{\langle 1 \rangle} q \cdot S + U_B^{\langle 1 \rangle} q + K^{\langle 1 \rangle} q) \cdot (\Delta M^{\langle 2 \rangle} + E^{\langle 2 \rangle} + U_A^{\langle 2 \rangle} q \cdot S + U_B^{\langle 2 \rangle} q + K^{\langle 2 \rangle} q) \bmod qbb_\alpha$

4. **$\mathsf{ModSwitch_{RNS}}$ from $qbb_\alpha \to bb_\alpha$:**
We switch the modulus of the ciphertext from qb to b by using $\mathsf{ModSwitch_{RNS}}$ as follows:
$$\left(\left\lceil \frac{t \cdot \hat{B}^{\langle 1 \rangle} \hat{B}^{\langle 2 \rangle}}{q} \right\rfloor, \left\lceil \frac{t \cdot \hat{A}^{\langle 1 \rangle} \hat{B}^{\langle 2 \rangle} + t \cdot \hat{A}^{\langle 2 \rangle} B^{\langle 1 \rangle}}{q} \right\rfloor, \left\lceil \frac{t \cdot \hat{A}^{\langle 1 \rangle} \hat{A}^{\langle 2 \rangle}}{q} \right\rfloor \right) \bmod bb_\alpha$$
, which is (almost, considering the rounding error) equivalent to a ciphertext encrypting the following plaintext:
$$\left\lceil \frac{t \cdot (\Delta M^{\langle 1 \rangle} + E^{\langle 1 \rangle} + U_A^{\langle 1 \rangle} q \cdot S + U_B^{\langle 1 \rangle} q + K^{\langle 1 \rangle} q) \cdot (\Delta M^{\langle 2 \rangle} + E^{\langle 2 \rangle} + U_A^{\langle 2 \rangle} q \cdot S + U_B^{\langle 2 \rangle} q + K^{\langle 2 \rangle} q)}{q} \right\rfloor \bmod bb_\alpha$$
$$= \left\lceil (M^{\langle 1 \rangle} + \frac{t}{q} \cdot E^{\langle 1 \rangle} + U_A^{\langle 1 \rangle} t \cdot S + U_B^{\langle 1 \rangle} t + K^{\langle 1 \rangle} t + \epsilon_d) \cdot (\Delta M^{\langle 2 \rangle} + E^{\langle 2 \rangle} + U_A^{\langle 2 \rangle} q \cdot S + U_B^{\langle 2 \rangle} q + K^{\langle 2 \rangle} q) \right\rfloor \bmod bb_\alpha$$
\# where $\epsilon_d = \frac{t}{q} \cdot \Delta M^{\langle 1 \rangle} - M^{\langle 1 \rangle}$ is a rounding error caused by treating $\frac{q}{t} \approx \left\lfloor \frac{q}{t} \right\rfloor = \Delta$

5. **$\mathsf{FastBConvEx_{RNS}}$ from $b \to q$:**
We exactly convert the base of the ciphertext from $bb_\alpha \to q$ as follows:
$$\left(\left\lceil \frac{t \cdot \hat{B}^{\langle 1 \rangle} \hat{B}^{\langle 2 \rangle}}{q} \right\rfloor, \left\lceil \frac{t \cdot \hat{A}^{\langle 1 \rangle} \hat{B}^{\langle 2 \rangle} + t \cdot \hat{A}^{\langle 2 \rangle} B^{\langle 1 \rangle}}{q} \right\rfloor, \left\lceil \frac{t \cdot \hat{A}^{\langle 1 \rangle} \hat{A}^{\langle 2 \rangle}}{q} \right\rfloor \right) \bmod q$$
, which is equivalent to a ciphertext encrypting the following plaintext:
$$= \left\lceil (M^{\langle 1 \rangle} + \frac{t}{q} \cdot E^{\langle 1 \rangle} + U_A^{\langle 1 \rangle} t \cdot S + U_B^{\langle 1 \rangle} t + K^{\langle 1 \rangle} t + \epsilon_d) \cdot (\Delta M^{\langle 2 \rangle} + E^{\langle 2 \rangle} + U_A^{\langle 2 \rangle} q \cdot S + U_B^{\langle 2 \rangle} q + K^{\langle 2 \rangle} q) \right\rfloor \bmod q$$
$$= \Big\lceil \Delta M^{\langle 1 \rangle} M^{\langle 2 \rangle} + \frac{t}{q} \cdot \Delta M^{\langle 2 \rangle} E^{\langle 1 \rangle} + U_A^{\langle 1 \rangle} t \Delta M^{\langle 2 \rangle} \cdot S + U_B^{\langle 1 \rangle} t \Delta M^{\langle 2 \rangle} + M^{\langle 1 \rangle} E^{\langle 2 \rangle} + \frac{t}{q} \cdot E^{\langle 1 \rangle} E^{\langle 2 \rangle}$$
$$+ U_A^{\langle 1 \rangle} E^{\langle 2 \rangle} t \cdot S + U_B^{\langle 1 \rangle} E^{\langle 2 \rangle} t + K^{\langle 1 \rangle} t \Delta M^{\langle 2 \rangle} + K^{\langle 1 \rangle} t E^{\langle 2 \rangle} + \epsilon_d \cdot (\Delta M^{\langle 2 \rangle} + E^{\langle 2 \rangle} + U_A^{\langle 2 \rangle} q \cdot S + U_B^{\langle 2 \rangle} q + K^{\langle 2 \rangle} q) \Big\rfloor \bmod q$$
$\approx \Delta M^{\langle 1 \rangle} M^{\langle 2 \rangle} \bmod q$ \# all other terms are relatively much smaller than $\Delta M^{\langle 1 \rangle} M^{\langle 2 \rangle}$ in modulo q

6. **Relinearization:**
Once we have derived the rescaled polynomial triple $(D'_0, D'_1, D'_2) \bmod q$, the final relinearization step is equivalent to deriving the synthetic ciphertexts ct_α and ct_β and then computing $\mathsf{ct}_\alpha + \mathsf{ct}_\beta$. ct_α is simply (D'_0, D'_1), and we can derive $\mathsf{ct}_\beta = \mathsf{RLWE}_{S,\sigma}(D_2 \cdot S^2)$ by using the $\mathsf{DecompMult_{RNS}}$ operation (Summary **??** in **??**). The final ciphertext-to-ciphertext addition of $\mathsf{ct}_\alpha + \mathsf{ct}_\beta$ can be performed by using regular RNS addition.

D-5.10.5 CKKS's Ciphertext-to-Ciphertext Multiplication

CKKS's ciphertext-to-ciphertext multiplication (Summary **??** in **??**) is almost the same as BFV's, except that CKKS does not need the $\mathsf{ModRaise}$ operation in the beginning (because each multi-

plicative level's modulus q_l is large enough to hold a multiplied scaled plaintext $\Delta^2 M^{\langle 1 \rangle} M^{\langle 2 \rangle}$). Therefore, CKKS's RNS-based multiplication is the same as BFV's except that it does not require step 1 (ModRaise), step 3 (constant multiplication by t), and step 5 (exact fast base conversion). Since a CKKS ciphertext's scaling factor Δ is approximately the same as the prime modulus factor of each multiplicative level, each ciphertext-to-ciphertext multiplication only needs to perform a modulus switch to a lower level.

D-5.10.6 BGV's Ciphertext-to-Ciphertext Multiplication

BGV's ciphertext-to-ciphertext multiplication (Summary **??** in **??**) is almost the same as CKKS's, except that BGV uses its own modulus switch ($\mathsf{ModSwitch}^{\mathsf{BGV}}_{\mathsf{RNS}}$ as described in Summary **??** in **??**) during the rescaling step. Therefore, BGV's RNS-based ciphertext-to-ciphertext multiplication is the same as CKKS's, except that $\mathsf{ModSwitch}_{\mathsf{RNS}}$ is replaced by $\mathsf{ModSwitch}^{\mathsf{BGV}}_{\mathsf{RNS}}$.

D-5.10.7 BFV's Bootstrapping

BFV's original bootstrapping procedure (Summary **??** in **??**) is as follows: (1) modulus switch from $q \to p^{\varepsilon}$; (2) homomorphic decryption; (3) CoeffToSlot; (4) EvalExp; (5) SlotToCoeff; and (6) re-interpretation.

However, in RNS, we cannot mod-switch to p^{ε} because RNS's base moduli have to be co-prime to each other, whereas the factors of p^{ε} are not. To avoid this issue, RNS-based BFV's bootstrapping instead performs the following: (1) $\mathsf{ModRaise}_{\mathsf{RNS}}$ from $q \to qbb_{\alpha}$, where bb_{α} is an auxiliary base; (2) coefficient multiplication by p^{ε}; (3) $\mathsf{ModSwitch}_{\mathsf{RNS}}$ from $qbb_{\alpha} \to bb_{\alpha}$; (4) FastBConvEx from $bb_{\alpha} \to q$; (5) homomorphic decryption to adjust the scaling factor of the plaintext; (6) CoeffToSlot; (7) EvalExp; (8) SlotToCoeff; and (9) re-interpretation. The detailed procedure is described as follows:

Input: The input BFV ciphertext to bootstrap is (A, B) mod q, which would decrypt to:
$A \cdot S + B = \Delta M + E + Kq$ # where $\Delta = \frac{q}{p^r}$

1. **$\mathsf{ModRaise}_{\mathsf{RNS}}$ from $q \to qbb_{\alpha}$:**
   Mod-raise ciphertext (A, B) mod q to (A, B) mod qbb_{α}, which would decrypt to:
   $A \cdot S + B = \Delta M + E + Kq + Uq \pmod{qbb_{\alpha}}$
   # where Uq is the FastBConv + SmallMont error, and U's coefficients are either $\{-1, 0, 1\}$

2. **Coefficient Multiplication by p^{ε}:**
   Multiply the coefficients of (A, B) mod q by p^{ε} to update the ciphertext to $(p^{\varepsilon}A, p^{\varepsilon}B)$ mod pbb_{α}, which would decrypt to:
   $p^{\varepsilon}A \cdot S + p^{\varepsilon}B = \Delta p^{\varepsilon}M + p^{\varepsilon}E + p^{\varepsilon}Kq + p^{\varepsilon}Uq \pmod{qbb_{\alpha}}$

3. **$\mathsf{ModSwitch}_{\mathsf{RNS}}$ from $qbb_{\alpha} \to bb_{\alpha}$:**
   Mod-switch the ciphertext $(p^{\varepsilon}A, p^{\varepsilon}B)$ mod pbb_{α} to $\left(\left\lceil \frac{p^{\varepsilon}A}{q} \right\rfloor, \left\lceil \frac{p^{\varepsilon}B}{q} \right\rfloor\right) \pmod{bb_{\alpha}}$, which would decrypt to:
   $\left\lceil \frac{p^{\varepsilon}A}{q} \right\rfloor \cdot S + \left\lceil \frac{p^{\varepsilon}B}{q} \right\rfloor = \frac{\Delta p^{\varepsilon}M}{q} + \frac{p^{\varepsilon}E}{q} + \frac{p^{\varepsilon}Kq}{q} + \frac{p^{\varepsilon}Uq}{q} + \epsilon \pmod{bb_{\alpha}}$
   # ϵ is a small rounding error

$= p^{\varepsilon-r}M + \dfrac{p^{\varepsilon}E}{q} + p^{\varepsilon}K + p^{\varepsilon}U + \epsilon \pmod{bb_\alpha}$

4. **FastBConvEx from $bb_\alpha \to q$:**
   Exact fast base conversion of $(p^{\varepsilon}A, p^{\varepsilon}B) \bmod bb_\alpha$ to $(p^{\varepsilon}A, p^{\varepsilon}B) \bmod q$, which would decrypt to: $p^{\varepsilon-r}M + \dfrac{p^{\varepsilon}E}{q} + p^{\varepsilon}K + p^{\varepsilon}U + \epsilon \bmod q$

5. **Homomorphic Decryption:**
   Now, we have the ciphertext $(p^{\varepsilon}A, p^{\varepsilon}B) \bmod q = \mathsf{RLWE}_{S,\sigma}\left(p^{\varepsilon-r}M + \dfrac{p^{\varepsilon}E}{q} + p^{\varepsilon}K + p^{\varepsilon}U + \epsilon\right) \bmod q$. We do homomorphic decryption by using the encrypted private key $\mathsf{RLWE}_{S,\sigma}(\hat{\Delta}S)$, where $\hat{\Delta} = \left\lfloor \dfrac{q}{p^{\varepsilon}} \right\rfloor$. The output is $\mathsf{RLWE}_{S,\sigma}(\hat{\Delta} \cdot (p^{\varepsilon-r}M + \dfrac{p^{\varepsilon}E}{q} + p^{\varepsilon}K + p^{\varepsilon}U + \epsilon)) \bmod q$.

6. Perform CoeffToSlot, digit extraction, and SlotToCoeff. These operations can be performed by only regular RNS-based additions and multiplications. The final digit-extracted ciphertext is $\mathsf{RLWE}_{S,\sigma}(\hat{\Delta} \cdot (p^{\varepsilon-r} \cdot M + Kp^{\varepsilon} + Up^{\varepsilon}))$, where all noise values smaller than the (base-p) $(\varepsilon - r)$-th digits are eliminated.

7. **Scaling Factor Re-interpretation:**
   Theoretically re-interpret the ciphertext without any additional mathematical computation. The ciphertext $\mathsf{RLWE}_{S,\sigma}(\hat{\Delta} \cdot (p^{\varepsilon-r} \cdot M + Kp^{\varepsilon} + Up^{\varepsilon}))$ is mathematically the same as:
   $\mathsf{RLWE}_{S,\sigma}(\hat{\Delta} \cdot (p^{\varepsilon-r} \cdot M + Kp^{\varepsilon} + Up^{\varepsilon}))$
   $= \mathsf{RLWE}_{S,\sigma}(\Delta M + (K+U) \cdot q)$ # since $\Delta = \dfrac{q}{p^r}$, and $\hat{\Delta} = \dfrac{a}{p^{\varepsilon}}$
   $= \mathsf{RLWE}_{S,\sigma}(\Delta M) \bmod q$.

D-5.10.8 CKKS's Bootstrapping

CKKS's original bootstrapping procedure (Summary **??** in **??**) is as follows: (1) Modraise; (2) homomorphic decryption; (3) CoeffToSlot; (4) EvalExp; (5) CoeffToSlot; (6) Re-interpretation. In the RNS-based CKKS bootstrapping, we perform $\mathsf{ModRaise}_{\mathsf{RNS}}$ at step 1, and all other steps are computed by using regular RNS-based addition and multiplication operations. Step 1's $\mathsf{ModRaise}_{\mathsf{RNS}}$ operation generates a $u \cdot q_0$ noise (where $u \in \{-1, 0, 1\}$ using SmallMont) for each polynomial coefficient during FastBConv. Therefore, step 2's homomorphic decryption outputs $\Delta M + E + W \cdot q_0 + K \cdot q_0$, where $W \cdot q_0$ represents the aggregation of all coefficient noise terms which are multiplied with the q_0-overflow noises generated by FastBConv and SmallMont. The $W \cdot q_0 + K \cdot q_0$ term gets eliminated by step 4's EvalExp which performs approximated modulo reduction based on a sine-graph evaluation whose period is q_0.

D-5.10.9 BGV's Bootstrapping

BGV's original bootstrapping procedure (**??**) is as follows: (1) modulus switch from $q_l \to \hat{q}$; (2) ciphertext coefficient multiplication by $p^{\varepsilon-1}$; (3) ModRaise; (4) CoeffToSlot; (5) EvalExp; (6) homomorphic multiplication by $|p^{-(\varepsilon-1)}|_{p^{\varepsilon}}$; (7) SlottToCoeff; (8) noise term re-interpretation. Given this procedure, the RNS-based bootstrapping steps are as follows:

Suppose the target BGV ciphertext to bootstrap is (A, B) mod q_l, where the plaintext modulus (i.e., noise scaling factor) is p.

1. $\mathsf{ModSwitch}^{\mathsf{BGV}}_{\mathsf{RNS}}$ from $q_l \to \hat{q}$ where $\hat{q}$ is a special modulus satisfying the following requirements: $\hat{q} \equiv 1 \bmod p^{\varepsilon}$, $\hat{q} \equiv 1 \bmod p$, $\hat{q}$ and q_l are co-prime, and $\hat{q} < q_l$.

2. **Constant multiplication** by $p^{\varepsilon-1}$ to the coefficients of the ciphertext polynomials $(\hat{A}, \hat{B})$, which increases the underlying plaintext's noise scaling factor Δ and the plaintext modulus from $p \to p^{\varepsilon}$. This effectively updates the underlying plaintext to $p^{\varepsilon-1}M + p^{\varepsilon}E$.

3. $\mathsf{ModRaise}_{\mathsf{RNS}}$ from $\hat{q} \to q_L$, which generates an additional noise $|u \cdot \hat{q}|_{q_L}$ (where $u \in \{-1, 0, 1\}$ using SmallMont). At this point, the ciphertext is $\mathsf{RLWE}_{S,\sigma}(p^{\varepsilon-1}M + p^{\varepsilon}E + \hat{q}K)$ mod q_L, whose underlying plaintext is:
   $p^{\varepsilon-1}M + p^{\varepsilon}E + \hat{q}K$
   $= p^{\varepsilon-1}M + K \bmod p^{\varepsilon}$ # since $\hat{q} \equiv 1 \bmod p^{\varepsilon}$

After the above steps, the remaining steps (i.e., CoeffToSlot, EvalExp, homomorphic multiplication by $|p^{-(\varepsilon-1)}|_{p^{\varepsilon}}$, SlotToCoeff, and re-interpretation) can be performed by regular RNS-based addition and multiplication operations.

D-5.10.10 Noise Impact of RNS Operations

When RNS techniques are used in FHE operations, the noise generated by $\mathsf{FastBConvEx}_{\mathsf{RNS}}$, $\mathsf{ModRaise}_{\mathsf{RNS}}$, and $\mathsf{ModSwitch}_{\mathsf{RNS}}$ is directly added to each coefficient of the ciphertext polynomials A and B. Since the decryption relation is $A \cdot S + B$, even the noise added to the coefficients of the polynomial A gets multiplied by a large factor due to the polynomial multiplication with S. Therefore, it is important to always ensure to reduce the generated noise of each $\mathsf{FastBConvEx}_{\mathsf{RNS}}$ by using it with SmallMont.

D-5.10.11 Python Source Code of RNS Primitives

We provide a Python script implementing the following exemplary RNS primitives: FastBConv, $\mathsf{ModRaise}_{\mathsf{RNS}}$, $\mathsf{ModDrop}_{\mathsf{RNS}}$, $\mathsf{ModSwitch}_{\mathsf{RNS}}$, SmallMont, and BaseBConvEx.

D-6 FHE Scheme Comparison and Summary

We summarize and compare TFHE, BFV, CKKS, and BGV as follows:

	Hard Problem Basis
TFHE	LWE
BFV **CKKS** **BGV**	RLWE

Table 7: Hard Problem Basis

	Unit Data Type
TFHE	Vector
BFV **CKKS** **BGV**	Polynomial

Table 8: Unit Data Type

	Plaintext
TFHE	Number $m \in \mathbb{Z}_t$ # t is a power of 2
BFV	Polynomial $M \in \mathbb{Z}_t[X]/X^n+1$ # t is a prime, and n is a power of 2
CKKS	Polynomial $M \in \mathbb{R}[X]/X^n+1$ # n is a power of 2
BGV	Polynomial $M \in \mathbb{Z}_t[X]/X^n+1$ # t is a prime, and n is a power of 2

Table 9: Plaintext

	Secret Key
TFHE	Vector $\vec{s} \xleftarrow{\$} \mathbb{Z}_2^k$ # \$ is a uniform random distribution
BFV **CKKS** **BGV**	Polynomial $S \xleftarrow{\$} \mathbb{Z}_3[X]/X^n+1$, where we set $\mathbb{Z}_3 = \{-1, 0, 1\}$

Table 10: Secret Key

	Ciphertext
TFHE	(Vector $\vec{a}$, Number b) $= (\vec{a} \xleftarrow{\$} \mathbb{Z}_q^k, \quad b \in \mathbb{Z}_q)$ # $q \gg t$, and t divides q
BFV **CKKS** **BGV**	(Polynomial A, B) $= (A \xleftarrow{\$} \mathbb{Z}_q[X]/X^n+1, \quad B \in \mathbb{Z}_q[X]/X^n+1)$ # $q \gg t$

Table 11: Ciphertext

	Noise
TFHE	Number $e \xleftarrow{\chi} \mathbb{Z}_q$ # χ is a Gaussian random distribution
BFV **CKKS** **BGV**	Polynomial $E \xleftarrow{\chi} \mathbb{Z}_q[X]/X^n+1$

Table 12: Noise

	Scaling Factor
TFHE	Used for Δm, where $\Delta = \frac{q}{t}$ # t divides q
BFV	Used for ΔM, where $\Delta = \lfloor \frac{q}{t} \rfloor$ # t is a prime
CKKS	Used for ΔM, where $\Delta \cdot \|\|M\|\|_\infty \ll q_0$ # q_0 is the lowest multiplicative level's ciphertext modulus
BGV	Used for ΔE, where $\Delta = t$ #t is a prime

Table 13: Scaling Factor

	Encryption
TFHE	$(\vec{a}, b)$ where $\vec{a} \xleftarrow{\$} \mathbb{Z}_q^k$, $b = \Delta m + e - a \cdot s \bmod q$, $e \xleftarrow{\chi} \mathbb{Z}_q$ # After using e each time, throw it away
BFV **CKKS**	(A, B) where $A \xleftarrow{\$} \mathbb{Z}_q[X]/(X^n+1)$, $B = \Delta M + E - A \cdot S \bmod q$, $E \xleftarrow{\chi} \mathbb{Z}_q[X]/(X^n+1)$ # After using E each time, throw it away
BGV	(A, B) where $A \xleftarrow{\$} \mathbb{Z}_q[X]/(X^n+1)$, $B = M + \Delta E - A \cdot S \bmod q$, $E \xleftarrow{\chi} \mathbb{Z}_q[X]/(X^n+1)$ # After using E each time, throw it away

Table 14: Encryption

	Cryptographic Relation
TFHE	$b + a \cdot s = \Delta m + e \pmod q$, where $\Delta = \frac{q}{t}$ # t divides q
BFV	$B + A \cdot S = \Delta M + E \pmod q$, where $\Delta = \lfloor \frac{q}{t} \rfloor$ # t is a prime
CKKS	$B + A \cdot S = \Delta M + E \pmod q$, where $\Delta \cdot \|\|M\|\|_\infty \ll q_0$ # q_0 is the lowest multiplicative level's ciphertext modulus
BGV	$B + A \cdot S = M + \Delta E \pmod q$, where $\Delta = t$ # t is a prime

Table 15: Cryptographic Relation

	Decryption Formula
TFHE	$m = \left\lceil \frac{(b + a \cdot s \bmod q)}{\Delta} \right\rfloor \bmod t$ $= \left\lceil \frac{(\Delta m + e)}{\Delta} \right\rfloor \bmod t$ # e gets eliminated if $e < \frac{\Delta}{2}$
BFV	$M = \left\lceil \frac{(B + A \cdot S \bmod q)}{\Delta} \right\rfloor \bmod t$ $= \left\lceil \frac{(\Delta M + E)}{\Delta} \right\rfloor \bmod t$ # E gets eliminated if $\|\|E\|\|_\infty < \frac{\Delta}{2}$
CKKS	$M = \left\lceil \frac{(B + A \cdot S \bmod q)}{\Delta} \right\rfloor_{\frac{1}{\Delta}}$ $= \left\lceil \frac{\Delta M + E}{\Delta} \right\rfloor_{\frac{1}{\Delta}}$ # The final noise remains as $\frac{E}{\Delta}$ (increase Δ to reduce it)
BGV	$M = (B + A \cdot S \bmod q) \bmod t$ $= (M + \Delta E) \bmod t$ # E gets removed if $\Delta E < q$

Table 16: Decryption Formula

	Ciphertext Modulus
TFHE	A single number q # $q \gg t$, and t divides q
BFV	A single number q # $q \gg t$
CKKS	An L-multiplicative-level modulus chain $\{q_0, q_1, \cdots, q_L\}$ # each $q_i = \prod_{j=0}^{l} w_j$, and each w_j is a CRT modulus having the property: $w_0 \gg \Delta \cdot \|\|M\|\|_\infty$, $w_j \approx \Delta$ (for $1 \leq j \leq L$)
BGV	An L-multiplicative-level modulus chain $\{q_0, q_1, \cdots, q_L\}$ # each $q_i = \prod_{j=0}^{l} w_j$, and each w_j is a CRT modulus having the property: $w_0 \equiv w_1 \equiv \cdots \equiv w_L \bmod t$

Table 17: Ciphertext Modulus

	Ciphertext-to-Ciphertext Addition
TFHE	- Ciphertext $\mathsf{LWE}_{\vec{s},\sigma}(\Delta m_1) = (\vec{a}_1, b_1) = (a_{1,0}, a_{1,1}, \cdots, a_{1,k-1}, b_1) \bmod q$ - Ciphertext $\mathsf{LWE}_{\vec{s},\sigma}(\Delta m_2) = (\vec{a}_2, b_2) = (a_{2,0}, a_{2,1}, \cdots, a_{2,k-1}, b_2) \bmod q$ $\mathsf{LWE}_{\vec{s},\sigma}(\Delta(m_1 + m_2)) = (\vec{a}_1 + \vec{a}_2, b_1 + b_2) \bmod q$
BFV	- Ciphertext $\mathsf{RLWE}_{S,\sigma}(\Delta M_1) = (A_1, B_1) \bmod q$ - Ciphertext $\mathsf{RLWE}_{S,\sigma}(\Delta M_2) = (A_2, B_2) \bmod q$ $\mathsf{RLWE}_{S,\sigma}(\Delta(M_1 + M_2)) = (A_1 + A_2, B_1 + B_2) \bmod q$
CKKS	- Ciphertext $\mathsf{RLWE}_{S,\sigma}(\Delta M_1) = (A_1, B_1) \bmod q_l$ - Ciphertext $\mathsf{RLWE}_{S,\sigma}(\Delta M_2) = (A_2, B_2) \bmod q_l$ $\mathsf{RLWE}_{S,\sigma}(\Delta(M_1 + M_2)) = (A_1 + A_2, B_1 + B_2) \bmod q_l$
BGV	- Ciphertext $\mathsf{RLWE}_{S,\sigma}(M_1) = (A_1, B_1) \bmod q_l$ - Ciphertext $\mathsf{RLWE}_{S,\sigma}(M_2) = (A_2, B_2) \bmod q_l$ $\mathsf{RLWE}_{S,\sigma}(M_1 + M_2) = (A_1 + A_2, B_1 + B_2) \bmod q_l$

Table 18: Ciphertext-to-Ciphertext Addition

	Ciphertext-to-Plaintext Addition
TFHE	- Ciphertext $\mathsf{LWE}_{\vec{s},\sigma}(\Delta m_1) = (\vec{a}_1, b_1) = (a_{1,0}, a_{1,1}, \cdots, a_{1,k-1}, b_1) \bmod q$ - Plaintext number $c \in \mathbb{Z}_t$ $\mathsf{LWE}_{\vec{s},\sigma}(\Delta(m_1 + c)) = (\vec{a}_1, b_1 + \Delta c) \bmod q$
BFV	- Ciphertext $\mathsf{RLWE}_{S,\sigma}(\Delta M_1) = (A_1, B_1) \bmod q$ - Plaintext polynomial $C \in \mathbb{Z}_t[X]/(X^n + 1)$ $\mathsf{RLWE}_{S,\sigma}(\Delta(M_1 + C)) = (A_1, B_1 + \Delta C) \bmod q$
CKKS	- Ciphertext $\mathsf{RLWE}_{S,\sigma}(\Delta M_1) = (A_1, B_1) \bmod q_l$ - Plaintext polynomial $C \in \mathbb{R}[X]/(X^n + 1)$ $\mathsf{RLWE}_{S,\sigma}(\Delta(M_1 + C)) = (A_1, B_1 + \Delta C) \bmod q_l$
BGV	- Ciphertext $\mathsf{RLWE}_{S,\sigma}(M_1) = (A_1, B_1) \bmod q_l$ - Plaintext polynomial $C \in \mathbb{Z}_t[X]/(X^n + 1)$ $\mathsf{RLWE}_{S,\sigma}(M_1 + C) = (A_1, B_1 + C) \bmod q_l$

Table 19: Ciphertext-to-Plaintext Addition

	Ciphertext-to-Plaintext Multiplication
TFHE	- Ciphertext $\mathsf{LWE}_{\vec{s},\sigma}(\Delta m_1) = (\vec{a}_1, b_1) = (a_{1,0}, a_{1,1}, \cdots, a_{1,k-1}, b_1) \bmod q$ - Plaintext number $c \in \mathbb{Z}_t$ $\mathsf{LWE}_{\vec{s},\sigma}(\Delta(m_1 \cdot c)) = (\vec{a}_1 \cdot c,\ b_1 \cdot c) \bmod q$
BFV	- Ciphertext $\mathsf{RLWE}_{S,\sigma}(\Delta M_1) = (A_1, B_1) \bmod q$ - Plaintext polynomial $C \in \mathbb{Z}_t[X]/(X^n + 1)$ $\mathsf{RLWE}_{S,\sigma}(\Delta(M_1 \cdot C)) = (A_1 \cdot C,\ B_1 \cdot C)$
CKKS	- Ciphertext $\mathsf{RLWE}_{S,\sigma}(\Delta M_1) = (A_1, B_1) \bmod q_l$ - Plaintext polynomial $C \in \mathbb{R}[X]/(X^n + 1)$ 1. Basic Multiplication $\mathsf{ct} = \mathsf{RLWE}_{S,\sigma}(\Delta^2(M_1 \cdot C)) = (A_1 \cdot \Delta C,\ B_1 \cdot \Delta C) \bmod q_l$ 2. Rescaling by $\frac{1}{\Delta}$: $\left\lceil \frac{\mathsf{ct}}{\Delta} \right\rfloor = \mathsf{RLWE}_{S,\sigma}(\Delta M_1 C) \bmod q_{l-1}$
BGV	- Ciphertext $\mathsf{RLWE}_{S,\sigma}(M_1) = (A_1, B_1) \bmod q_l$ - Plaintext polynomial $C \in \mathbb{Z}_t[X]/(X^n + 1)$ $\mathsf{RLWE}_{S,\sigma}(M_1 \cdot C) = (A_1 \cdot C,\ B_1 \cdot C) \bmod q_l$

Table 20: Ciphertext-to-Plaintext Multiplication

	Ciphertext-to-Ciphertext Multiplication
TFHE	- Ciphertext $\mathsf{LWE}_{\vec{s},\sigma}(\Delta m_1) = (\vec{a}_1, b_1) = (a_{1,0}, a_{1,1}, \cdots, a_{1,k-1}, b_1) \bmod q$ - Ciphertext $\mathsf{LWE}_{\vec{s},\sigma}(\Delta m_2) = (\vec{a}_2, b_2) = (a_{2,0}, a_{2,1}, \cdots, a_{2,k-1}, b_2) \bmod q$ 1. Programmable Bootstrapping: Convert $\mathsf{LWE}_{\vec{s},\sigma}(\Delta m_2)$ into $\mathsf{GSW}^{\beta,l}_{\vec{s},\sigma}(m_2)$. 2. Homomorphic Multiplication: Compute $\mathsf{LWE}_{\vec{s},\sigma}(\Delta m_1) \cdot \mathsf{GSW}^{\beta,l}_{\vec{s},\sigma}(m_2)$ $= \sum_{i=0}^{k-1} \langle \mathsf{Decomp}^{\beta,l}(a_{1,i}), \mathsf{Lev}^{\beta,l}_{\vec{s},\sigma}(-s_i \cdot m_2) \rangle + \langle \mathsf{Decomp}^{\beta,l}(b_1), \mathsf{Lev}^{\beta,l}_{\vec{s},\sigma}(m_2) \rangle$ $= \mathsf{LWE}_{\vec{s},\sigma}(\Delta m_1 m_2)$
BFV	- Ciphertext $\mathsf{RLWE}_{S,\sigma}(\Delta M_1) = (A_1, B_1) \bmod q$ - Ciphertext $\mathsf{RLWE}_{S,\sigma}(\Delta M_2) = (A_2, B_2) \bmod q$ 1. ModRaise from q to $Q = q \cdot \Delta$ - Ciphertext $\mathsf{RLWE}_{S,\sigma}(\Delta M_1) = (A_1, B_1) \bmod Q$ - Ciphertext $\mathsf{RLWE}_{S,\sigma}(\Delta M_2) = (A_2, B_2) \bmod Q$ 2. Polynomial Multiplication: $(A_1A_2,\ A_1B_2 + A_2B_1,\ B_1B_2) \equiv (D_0, D_1, D_2) \pmod{Q}$ 3. Relinearization: $\mathsf{ct}_\alpha = (D_1, D_0),\ \mathsf{ct}_\beta = \langle \mathsf{Decomp}^{\beta,l}(D_2),\ \mathsf{RLev}^{\beta,l}_{S,\sigma}(S^2) \rangle$ $\mathsf{ct}_\alpha + \mathsf{ct}_\beta = \mathsf{ct}_{\alpha+\beta} = \mathsf{RLWE}_{S,\sigma}(\Delta^2 M_1 M_2) \bmod Q$ 4. Rescaling by $\frac{1}{\Delta}$: $\left\lceil \frac{\mathsf{ct}_{\alpha+\beta}}{\Delta} \right\rfloor = \mathsf{RLWE}_{S,\sigma}(\Delta M_1 M_2) \bmod q$
CKKS	- Ciphertext $\mathsf{RLWE}_{S,\sigma}(\Delta M_1) = (A_1, B_1) \bmod q_l$ - Ciphertext $\mathsf{RLWE}_{S,\sigma}(\Delta M_2) = (A_2, B_2) \bmod q_l$ 1. Polynomial Multiplication: $(A_1A_2,\ A_1B_2 + A_2B_1,\ B_1B_2) \equiv (D_0, D_1, D_2) \pmod{q_l}$ 2. Relinearization: $\mathsf{ct}_\alpha = (D_1, D_0),\ \mathsf{ct}_\beta = \langle \mathsf{Decomp}^{\beta,l}(D_2),\ \mathsf{RLev}^{\beta,l}_{S,\sigma}(S^2) \rangle$ $\mathsf{ct}_\alpha + \mathsf{ct}_\beta = \mathsf{ct}_{\alpha+\beta} = \mathsf{RLWE}_{S,\sigma}(\Delta^2 M_1 M_2) \bmod q_l$ 3. Rescaling by $\frac{1}{\Delta}$: $\left\lceil \frac{\mathsf{ct}_{\alpha+\beta}}{\Delta} \right\rfloor = \mathsf{RLWE}_{S,\sigma}(\Delta M_1 M_2) \bmod q_{l-1}$
BGV	- Ciphertext $\mathsf{RLWE}_{S,\sigma}(M_1) = (A_1, B_1) \bmod q_l$ - Ciphertext $\mathsf{RLWE}_{S,\sigma}(M_2) = (A_2, B_2) \bmod q_l$ 1. Polynomial Multiplication: $(A_1A_2,\ A_1B_2 + A_2B_1,\ B_1B_2) \equiv (D_0, D_1, D_2) \pmod{q_l}$ 2. Relinearization: $\mathsf{ct}_\alpha = (D_1, D_0),\ \mathsf{ct}_\beta = \langle \mathsf{Decomp}^{\beta,l}(D_2),\ \mathsf{RLev}^{\beta,l}_{S,\sigma}(S^2) \rangle$ $\mathsf{ct}_\alpha + \mathsf{ct}_\beta = \mathsf{ct}_{\alpha+\beta} = \mathsf{RLWE}_{S,\sigma}(M_1 M_2) \bmod q_l$ 3. (Optional) Rescaling by $\frac{1}{\Delta}$: $\left\lceil \frac{\mathsf{ct}_{\alpha+\beta}}{\Delta} \right\rfloor_t = \mathsf{RLWE}_{S,\sigma}(M_1 M_2) \bmod q_{l-1}$ # $\lceil \rfloor_t$ means rounding to the nearest multiple of t # The future noise growth rate gets reduced if the ciphertext is rescaled

Table 21: Ciphertext-to-Ciphertext Multiplication

	Maximum Possible Multiplications (without Bootstrapping)
TFHE	Unlimited with programming bootstrapping (but not possible without it)
BFV	Unlimited
CKKS	As many times as the length of the modulus chain
BGV	As many times as the length of the modulus chain

Table 22: Maximum Possible Multiplications (without Bootstrapping)

	Key Switching
TFHE	Key-switching from $\vec{s} \to \vec{s}'$: $\mathsf{LWE}_{\vec{s}',\sigma}(\Delta m) = b + a \cdot \mathsf{LWE}_{\vec{s}',\sigma}(s)$ $= b + \langle \mathsf{Decomp}^{\beta,l}(\vec{a}),\ \mathsf{Lev}^{\beta,l}_{\vec{s}',\sigma}(\vec{s}) \rangle$
BFV **CKKS**	Key-switching from $S \to S'$: $\mathsf{RLWE}_{S',\sigma}(\Delta M) = B + A \cdot \mathsf{RLWE}_{S',\sigma}(S)$ $= B + \langle \mathsf{Decomp}^{\beta,l}(A),\ \mathsf{RLev}^{\beta,l}_{S',\sigma}(S) \rangle$
BGV	Key-switching from $S \to S'$: $\mathsf{RLWE}_{S',\sigma}(M) = B + A \cdot \mathsf{RLWE}_{S',\sigma}(S)$ $= B + \langle \mathsf{Decomp}^{\beta,l}(A),\ \mathsf{RLev}^{\beta,l}_{S',\sigma}(S) \rangle$

Table 23: Key Switching

	Modulus Drop (**ModDrop**)
CKKS **BGV**	- Ciphertext with the multiplicative level l: $\mathsf{RLWE}_{S,\sigma}(\Delta M) = (A, B) \bmod q_l$ - Ciphertext with the multiplicative level $l-1$: $\mathsf{RLWE}_{S,\sigma}(\Delta M) = (A', B') = (A \bmod q_{l-1}, B \bmod q_{l-1})$

Table 24: Modulus Drop (ModDrop)

	Encoding and Decoding the Plaintext
TFHE	No need, because each plaintext is a single number
BFV **CKKS** **BGV**	Must convert the input slots into polynomial coefficients to support batch processing: - Encoding input slots $\vec{v}$ into polynomial coefficients: $\vec{m} = n^{-1} \cdot \vec{v} \cdot I_n^R \cdot \tilde{W}$ - Decoding polynomial coefficients $\vec{m}$ into input slots: $\vec{v} = \vec{m} \cdot \tilde{W}^*$

Table 25: Encoding and Decoding the Plaintext

	Input Slot Rotation
TFHE	Not applicable, because its plaintext is a single number (i.e., a single slot)
BFV **CKKS**	Given $\mathsf{ct} = \mathsf{RLWE}_{S(X),\sigma}(\Delta M(X)) = (A(X), B(X))$, to rotate the input slots by h positions to the left: 1. Update ct to $\mathsf{RLWE}_{S(X^{J(h)}),\sigma}(\Delta M(X^{J(h)})) = (A(X^{J(h)}),\ B(X^{J(h)}))$ (where $J(h) = 5^h \bmod 2n$) 2. Key-switch $\mathsf{RLWE}_{S(X^{J(h)}),\sigma}(\Delta M(X^{J(h)}))$ to $\mathsf{RLWE}_{S(X),\sigma}(\Delta M(X^{J(h)}))$.
BGV	Given $\mathsf{ct} = \mathsf{RLWE}_{S(X),\sigma}(M(X)) = (A(X), B(X))$, to rotate the input slots by h positions to the left: 1. Update ct to $\mathsf{RLWE}_{S(X^{J(h)}),\sigma}(M(X^{J(h)})) = (A(X^{J(h)}),\ B(X^{J(h)}))$ 2. Key-switch $\mathsf{RLWE}_{S(X^{J(h)}),\sigma}(M(X^{J(h)}))$ to $\mathsf{RLWE}_{S(X),\sigma}(M(X^{J(h)}))$.

Table 26: Input Slot Rotation

	Bootstrapping Goal
TFHE **BFV**	To reset the noise.
CKKS **BGV**	To reset the ciphertext modulus from $q_0 \to q_L$.

Table 27: Bootstrapping Goal

	Bootstrapping Details
TFHE	1. Modulus Switch from $q \to 2n$ to convert $\mathsf{LWE}_{\vec{s},\sigma}(\Delta m) \to \mathsf{LWE}_{\vec{s},\sigma}(\hat{\Delta} m) = (\vec{\hat{a}}, \hat{b}) \bmod 2n$, where $\hat{\Delta} = \frac{2n}{t}$. # where t divides $2n$ 2. Blind Rotation: Homomorphically rotate the GLWE-encrypted look-up table polynomial $\mathsf{GLWE}_{\vec{S},\sigma}(\Delta V)$ by $\Delta m + e$ positions to the left. This is done by by homomorphically deriving $\mathsf{GLWE}_{\vec{S},\sigma}(\Delta V_k)$ as follows: $\mathsf{GLWE}_{\vec{S},\sigma}(\Delta V_0) = \mathsf{GLWE}_{\vec{S},\sigma}(\Delta V) \cdot X^{-\hat{b}}$ $\mathsf{GLWE}_{\vec{S},\sigma}(\Delta V_i) = \mathsf{GLWE}_{\vec{S},\sigma}(\Delta V_{i-1}) \cdot X^{\hat{a}_i s_{i-1}}$ $= \mathsf{GGSW}^{\beta,l}_{\vec{S},\sigma}(s_{i-1}) \cdot (\mathsf{GLWE}_{\vec{S},\sigma}(\Delta V_{i-1}) \cdot X^{\hat{a}_{i-1}} - \mathsf{GLWE}_{\vec{S},\sigma}(\Delta V_{i-1})) + \mathsf{GLWE}_{\vec{S},\sigma}(\Delta V_{i-1})$ 3. Coefficient Extraction: The rotated encrypted polynomial V_k's constant term value is Δm. Extract this encrypted constant term as $\mathsf{LWE}_{\vec{s},\sigma}(\Delta m)$ from $\mathsf{GLWE}_{\vec{S},\sigma}(\Delta V_k)$.
BFV	1. Modulus Switch from $q \to p^\varepsilon$ to convert $\mathsf{RLWE}_{S,\sigma}(\Delta M) \to \mathsf{RLWE}_{S,\sigma}(p^{\varepsilon-1} M) \bmod p^\varepsilon$ 2. Homomorphic Decryption: $B + A \cdot \mathsf{RLWE}_{S,\sigma}(\Delta' S) = \mathsf{RLWE}_{S,\sigma}(\Delta' \cdot (p^{\varepsilon-1} M + E + K p^\varepsilon)) \bmod q$, where $\Delta' = \left\lfloor \frac{q}{p^\varepsilon} \right\rfloor$ 3. CoeffToSlot: Multiply to the ciphertext by $n^{-1} \cdot I_n^R \cdot \tilde{W}$ to move the plaintext coefficients of $p^{\varepsilon-1} M + E + K p^\varepsilon$ to the input slots. 4. EvalExp: Given the digit extraction polynomial $G_{\varepsilon,v}(x)$, homomorphically compute: $G_{\varepsilon,1} \circ G_{\varepsilon,2} \circ \cdots \circ G_{\varepsilon,w-1}(p^{\varepsilon-1} M + E + K p^\varepsilon)$, and then the output $p^{\varepsilon-1} M + K' p^\varepsilon$ gets stored in the plaintext slots. 5. SlotToCoeff: Multiply to the ciphertext by $\tilde{W}^*$ to move $p^{\varepsilon-1} M + K' p^\varepsilon$ to the polynomial coefficient positions and get $\mathsf{RLWE}_{S,\sigma}(\Delta' \cdot (p^{\varepsilon-1} M + K' p^\varepsilon)) \bmod q$. 6. Scaling Factor Re-interpretation: View $\mathsf{RLWE}_{S,\sigma}(\Delta' \cdot (p^{\varepsilon-1} M + K' p^\varepsilon)) \bmod q$ as $\mathsf{RLWE}_{S,\sigma}\left(\left\lfloor \frac{q}{p^\varepsilon} \right\rfloor \cdot (p^{\varepsilon-1} M + K' p^\varepsilon)\right) \approx \mathsf{RLWE}_{S,\sigma}\left(\left\lfloor \frac{q}{p} \right\rfloor \cdot (M) + K' q\right)$ $= \mathsf{RLWE}_{S,\sigma}(\Delta M) \bmod q$, an encryption of M with the scaling factor $\Delta = \left\lfloor \frac{q}{p} \right\rfloor$

Table 28: Bootstrapping Details: TFHE, BFV

	Bootstrapping Details
CKKS	1. ModRaise: View the ciphertext (A, B) mod q_0 as (A, B) mod q_L 2. CoeffToSlot: Move the coefficients of $\Delta M + E + Kq_0$ to the input slots. 3. EvalExp: Homomorphically evaluate the polynomial approximation of the sine function with period q_0 at $\Delta M + E + Kq_0$, which outputs an encryption of $\Delta M + E$ in the plaintext slots. 4. SlotToCoeff: Move $\Delta M + E$ to the polynomial coefficient positions to get $\mathsf{RLWE}_{S,\sigma}(\Delta M + E)$.
BGV	1. Modulus Switch from $q_l \to \hat{q}$ to convert $\mathsf{RLWE}_{S,\sigma}(M) = (A, B)$ mod $q_0 \to \mathsf{RLWE}_{S,\sigma}(M) = (\hat{A}, \hat{B})$ mod $\hat{q}$, where $\hat{q} \equiv 1 \bmod p^{\varepsilon}$ 2. Ciphertext Coefficient Multiplication by $p^{\varepsilon-1}$: Compute $(p^{\varepsilon-1}\hat{A}, p^{\varepsilon-1}\hat{B}) = (A', B')$ mod $\hat{q}$ (where $\hat{q} \equiv 1 \bmod p^{\varepsilon}$), which the ciphertext $\mathsf{RLWE}_{S,\sigma}(p^{\varepsilon-1}M)$ mod $\hat{q}$ with noise $p^{\varepsilon}E$. 3. ModRaise: (A', B') mod $\hat{q} \to (A', B')$ mod q_L, which is the ciphertext $\mathsf{RLWE}_{S,\sigma}(p^{\varepsilon-1}M + p^{\varepsilon}E + K\hat{q})$ mod q_L. 4. CoeffToSlot: Multiply to the ciphertext by $n^{-1} \cdot I_n^R \cdot \tilde{W}$ to move the plaintext coefficients of $p^{\varepsilon-1}M + p^{\varepsilon}E + K\hat{q}$ to the input slots. 5. EvalExp: Given the digit extraction polynomial $G_{w,v}(x)$, homomorphically compute: $\big(\, G_{w,1} \circ G_{w,2} \circ \cdots \circ G_{w,w-1}(p^{\varepsilon-1}M + p^{\varepsilon}E + K\hat{q}) \,\big) \cdot p^{\varepsilon-1}$, and then the output $Mp^{\varepsilon-1} + K'p^{\varepsilon}$ gets stored in the plaintext slots. 6. Homomorphic Multiplication by $\lvert p^{-(\varepsilon-1)} \rvert_{p^{\varepsilon}}$ to all slots to update the plaintext from $Mp^{\varepsilon-1} + K'p^{\varepsilon} \pmod{p^{\varepsilon}}$ to $M + K'p \pmod{p^{\varepsilon}}$. 7. SlotToCoeff: Multiply to the ciphertext by $\tilde{W}^*$ to move $M + K'p$ to the polynomial coefficient positions to get $\mathsf{RLWE}_{S,\sigma}(M + K'p)$ mod q_L. 8. Noise Term Re-interpretation: View $\mathsf{RLWE}_{S,\sigma}(M + K'p)$ mod q_L as $\mathsf{RLWE}_{S,\sigma}(M)$ mod q_L, an encryption of M with noise $E' = K'$ having the noise scaling factor (i.e., plaintext modulus) $\Delta = p$.

Table 29: Bootstrapping Details: CKKS, BGV

	Noise Management
TFHE **BFV**	Their bootstrapping resets the noise.
CKKS	- The noise grows without stopping, because its bootstrapping resets only the modulus chain. To slow down the noise growth, we should increase the plaintext's scaling factor Δ. - CKKS's EvalExp cannot use the digit extraction polynomial to remove the noise, because CKKS's plaintext is not in a modulus ring, but is a real number.
BGV	BGV's modulus switch has the special property of resetting the noise, and BGV's bootstrapping resets the modulus chain to enable indefinite modulus switches.

Table 30: Noise Management