A STUDY ON CRYSTALS-KYBER AND ITS MASKED IMPLEMENTATIONS

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF APPLIED MATHEMATICS
OF
MIDDLE EAST TECHNICAL UNIVERSITY

BY

SILA ÖZEREN

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR
THE DEGREE OF MASTER OF SCIENCE
IN
CRYPTOGRAPHY

SEPTEMBER 2023

Approval of the thesis:

A STUDY ON CRYSTALS-KYBER AND ITS MASKED IMPLEMENTATIONS

submitted by **SILA ÖZEREN** in partial fulfillment of the requirements for the degree of **Master of Science in Cryptography Department, Middle East Technical University** by,

Prof. Dr. A. Sevtap Selçuk Kestel ______________
Dean, Graduate School of **Applied Mathematics**

Assoc. Prof. Dr. Oğuz Yayla ______________
Head of Department, **Cryptography**

Assoc. Prof. Dr. Oğuz Yayla ______________
Supervisor, **Cryptography, METU**

Examining Committee Members:

Prof. Dr. Zülfükar Saygı
Mathematics, TOBB University of Economics and Technology ______________

Assoc. Prof. Dr. Oğuz Yayla
Cryptography, Middle East Technical University ______________

Prof. Dr. Barış Bülent Kırlar
Mathematics, Süleyman Demirel University ______________

Assist. Prof. Dr. Buket Özkaya
Cryptography, Middle East Technical University ______________

Assist. Prof. Dr. Erdem Alkım
Computer Science, Dokuz Eylül University ______________

Date: ______________

I hereby declare that all information in this document has been obtained and presented in accordance with academic rules and ethical conduct. I also declare that, as required by these rules and conduct, I have fully cited and referenced all material and results that are not original to this work.

Name, Last Name: SILA ÖZEREN

Signature :

ABSTRACT

A STUDY ON CRYSTALS-KYBER AND ITS MASKED IMPLEMENTATIONS

Özeren, Sıla
M.S., Department of Cryptography
Supervisor : Assoc. Prof. Dr. Oğuz Yayla

September 2023, 128 pages

As we transition into the quantum computing era, the security of widely-used cryptographic algorithms is facing significant challenges. This is attributable to Shor's algorithm, enabling quantum computers to break conventional cryptosystems such as RSA, DSA, and elliptic curve cryptosystems. This thesis provides a comprehensive study on the CRYSTALS-Kyber key encapsulation mechanism (KEM), the only KEM algorithm that was a third-round finalist in NIST's PQC Standardization effort. We begin with a detailed examination of the foundational concepts of lattices, introducing the inherent hard problems in lattice cryptography, including Learning with Errors (LWE), Ring-LWE, and Module-LWE. We subsequently delve into the three components of Kyber.CPAPKE and detail the Fujisaki-Okamoto transform version of each algorithm necessary to achieve IND-CCA2 security. An extensive study is conducted on existing masking methods for the compression function in Kyber, and their shortcomings due to prime modulo design are highlighted. We propose two methods for masking this compression function: one integrating a look-up-table, and the other utilizing a double-and-check method. Additionally, we introduce potential compression functions for various prime numbers.

Keywords: post-quantum cryptography, lattice cryptography, CRYSTALS-KYBER, key encapsulation mechanism, side-channel attacks, masking countermeasures

ÖZ

CRYSTALS-KYBER VE MASKELENMİŞ UYGULAMALARI ÜZERİNE BİR ÇALIŞMA

Özeren, Sıla
Yüksek Lisans, Kriptografi Bölümü
Tez Yöneticisi : Doç. Dr. Oğuz Yayla

Eylül 2023, 128 sayfa

Kuantum hesaplama çağına geçiş yaparken, RSA, DSA ve eliptik eğri kriptosistemleri gibi birçok yaygın kullanılan kriptografik algoritmanın güvenliği önemli zorluklarla karşı karşıya kalıyor. Bu tezde, NIST'in kuantum-sonrası kriptografi standardizasyon sürecinin finalistlerinden biri olan CRYSTALS-Kyber anahtar kapsülleme mekanizması üzerine kapsamlı bir inceleme sunmaktayız. Kafes-tabanlı kriptografinin temel kavramlarının detaylı bir açıklamasıyla başlayarak, bu alanda bilinen zor problemleri tanıtıyoruz. Daha sonra, Kyber.CPAPKE'nin üç bileşenini detaylıca inceliyoruz ve her bir algoritmanın IND-CCA2 güvenliği için gerekli olan Fujisaki-Okamoto dönüşüm versiyonlarını sunuyoruz. Kyber'deki kompresyon fonksiyonu için mevcut maskeleme yöntemleri üzerine detaylı bir çalışma yürütüyoruz ve bunların asal modül tasarımı nedeniyle birtakım niteliklerden yoksun olduğunu belirtiyoruz. Bu kompresyon fonksiyonunu maskelemek için iki yöntem öneriyoruz. Biri bir arama tablosunu entegre ederken, diğeri bir çiftle-ve-kontrol et yöntemini kullanıyor. Ek olarak, çeşitli asal sayılar için potansiyel kompresyon fonksiyonlarını sunuyoruz.

Anahtar Kelimeler: kuantum sonrası kriptografi, kafes-tabanlı kriptografi, CRYSTAL-Kyber, anahtar kapsülleme mekanizması, yan kanal saldırıları, maskeleme karşı önlemleri

To my cherished cat, Gülben, a deeply missed member of our family, whose warm embraces and charming round face, outlined by naturally kohled green eyes, continue to grace my dreams even after all these years.

ACKNOWLEDGMENTS

I would like to express my utmost gratitude to my supervisor, Assoc. Prof. Dr. Oğuz Yayla. His guidance and expertise have been invaluable throughout my master's program.

I am also deeply grateful to TUBİTAK for their support during the challenging times of the COVID-19 pandemic through the "2210-A National MSc/MA Scholarship Program". Their financial assistance has been crucial in enabling me to sustain myself during especially the first year of my master's program.

I would like to extend my heartfelt thanks to my parents Abdurrahman Özeren and Yıldız Özeren for their unwavering love and patience. Their constant support has played a pivotal role in completing this thesis.

I'd like to extend my heartfelt thanks to Yasemin Sipahi. Even though you were nearly 10 thousand km away from me, you were right there with me in spirit. I'm especially reminded of you by the ring we bought from that eccentric woman with a shop in Kızılay.

I would also like to express my profound gratitude to Melike Çakmak. Her unwavering support has been a constant source of strength throughout this journey. In moments of frustration and challenge, she has been my rock, providing an empathetic shoulder to lean on. Her presence has made this journey more manageable and I am deeply thankful for her support.

I would like to extend my heartfelt gratitude to Dr. Süleyman Özarslan for his invaluable guidance, profound expertise, and unwavering patience during our interactions at the office.

Finally, I express my deepest gratitude to my *significant other*, Gökhan Hacıoğlu. His steadfast devotion and continual encouragement have been the bedrock upon which my success has been built. Even when I grappled with self-doubt, burdened by my high expectations, he never faltered in his belief in me. He saw the best in me when I was at my worst, and his unwavering faith propelled me forward even when the journey seemed unbearable. His love and support served as an enduring source of motivation, inspiring me to persevere through every challenge and complete my thesis.

TABLE OF CONTENTS

ABSTRACT vii

ÖZ ix

ACKNOWLEDGMENTS xi

TABLE OF CONTENTS xiii

LIST OF TABLES xix

LIST OF FIGURES xx

LIST OF ABBREVIATIONS xxii

CHAPTERS

1 INTRODUCTION 1

2 INTRODUCTION TO LATTICE-BASED CRYPTOGRAPHY 7

2.1 Introductory Definitions and Theorems for Lattice-based Cryptography 7

2.2 Lattice-based Hard Problems 9

2.3 Short Integer Solution (SIS) Hard Problem 10

2.4 Learning With Error (LWE) Hard Problem 11

2.4.1 Matrix Representation of the Learning-With-Error (LWE) Problem 13

2.5 The Ring Learning With Error Problem (Ring-LWE) 14
2.5.1 Search Ring Learning With Error Problem 15
2.5.2 Decision Ring Learning With Error Problem 16
2.5.3 Matrix Representation of the Ring Learning With Error Problem 16
2.6 Module Learning with Errors Problem (Module-LWE) 18
2.6.1 The Hardness of the Module Learning with Errors . 19
2.6.2 Matrix Representation of the Module-LWE 21
3 TECHNICAL BACKGROUND 23
3.1 Indistinguishability Under Chosen Plaintext Attacks (IND-CPA) 23
3.2 Indistinguishability Under (Adaptive) Chosen Ciphertext Attacks (IND-CCA) 24
3.3 Masking: Dividing a Sensitive Value into Uniformly Selected Shares 25
3.3.1 Need for Masking 25
3.3.2 Circuits and Gadgets They Are Working on 26
3.3.3 Arithmetic Encoding of a Sensitive Variable $x \in \mathbb{Z}_q$ 26
3.3.3.1 A Toy Example for an Arithmetic Encoding 27
3.3.4 Boolean Encoding of a Sensitive Variable $x \in \mathbb{Z}_2^k$. 28
3.3.4.1 A Toy Example for Boolean Encoding of a Sensitive Variable $x \in \mathbb{Z}_{2^k}$ . . 28
3.3.5 Security Definitions for Shared Implementations . 29

3.3.5.1 t-NI Security Notion 30

3.3.5.2 t-SNI Security Notion 30

4 CRYSYALS-KYBER KEY ENCAPSULATION MECHANISM (KEM) 33

4.1 Parameters to be Used 33

4.2 Notations and Background Knowledge for the CPAKEM and CCAKEM Secure CRYSTAL-Kyber Algorithms 35

4.2.1 Byte and Byte Arrays 35

4.2.2 The Polynomial Ring $\mathbb{R}_q$ Represented as $\mathbb{Z}_q[X]/(X^n+1)$ 38

4.2.3 Modular Reduction 40

4.2.4 Rounding 40

4.2.5 Compression and Decompression Functions 40

4.2.5.1 A Toy Example for the Compression Function 41

4.2.5.2 Motivations Behind Compression and Decompression Functions 42

4.2.6 Encoding and Decoding Functions 43

4.2.7 Symmetric Primitives 44

4.2.8 Number-Theoretic Transform (NTT) 44

4.2.8.1 The 256th Roots of Unity for the Defining Polynomial $X^{256}+1$ 44

4.2.8.2 How to Prove that the Defining Polynomial Has 256th Roots of Unity? . . 45

4.2.8.3 Defining Polynomial $X^{256}+1$ Factors into 128 Quadratic Polynomials 45

4.2.8.4 Performing NTT on Polynomial $f \in \mathbb{R}_q$: Representation as 128 First-Degree Polynomials 46

4.2.8.5 Efficient Multiplication of Two Polynomails with NTT. 49

4.2.9 Uniform Sampling From the Ring of Polynomials $\mathbb{R}_q$ 49

4.2.10 Sampling from a Binomial Distribution 50

4.3 Kyber.CPAPKE.KeyGen() Algorithm 51

4.3.1 Length of the Secret and Public Keys 56

4.4 Kyber.CPAPKE.Enc(pk, m, r) 57

4.5 Kyber.CPAPKE.Dec(c, sk) 63

4.6 Kyber.CCAKEM.KeyGen() 64

4.6.1 Length of the Secret and Public Keys 65

4.7 Kyber.CCAKEM.Encaps(pk, m, r) 66

4.8 Kyber.CCAKEM.Enc(pk) 66

4.9 Kyber.CCAKEM.Decaps(c, sk) 68

4.9.1 Explanation of the Length of Certain Algorithmic Components 71

4.9.2 Leakage Risk Point of the Algorithm 72

5 DEEP DIVE INTO THE HIGHER-ORDER ONE-BIT COMPRESSION ALGORITHM 75

5.1 Introduction to the Algorithm 75

5.2 Unmasked Compression in Kyber.CPAPKE.Dec() Algorithm 77

5.2.1 Toy Example for Unmasked Compression Function 79

5.3 Trivial Masking Approach for Algorithms that Uses Power-of-Two Modulo Like Saber 79

5.4 Higher-Order One-Bit Compression 82

5.4.1 The Case 1: Where the First Bit is Set to 1 $(x_{11} = 1)$ 85

5.4.2 The Case 2: Where $(x_{11} = 0, x_{10} = 0)$ 86

5.4.3 The Case 3: Where $(x_{11} = 0, x_{10} = 1, x_9 = 0)$ . . 86

5.4.4 The Case 4: Where $(x_{11} = 0, x_{10} = 1, x_9 = 1, x_8 = 1)$ 87

5.4.5 The Case 5: Where $(x_{10} = 1, x_9 = 1, x_8 = 0, x_7 = 1)$ 88

5.5 Probing Security of the Algorithm 13 90

5.5.1 Gadgets: The Fundamental Units in Cryptographic Systems 91

5.5.2 Gadgets Employed in the Higher-Order One-Bit Compression Algorithm 13 91

5.5.3 t-SNI Security of the Algorithm 13 93

5.5.4 Proof of the Theorem 1 [14] 94

6 ALTERNATIVE MASKING 99

6.1 Introduction 99

6.2 The Double and Check Compression Method 100

6.2.1 Pseudo-code of the Double and Check Algorithm . 101

6.2.2 Toy Example for the Double and Check Algorithm 102

6.2.3 Final Notes On the Double and Check Method . . 103

6.3 Look-Up-Table (LUT) Based Compression Algorithm (32 Entities) . . . 105

6.3.1 Pseudo-code of the Look-Up-Table (LUT) Based Compression Algorithm . . . 107

6.3.2 Toy Example for the Look-Up-Table (LUT) Based Offset and Check Algorithm . . . 109

6.3.3 Notes On the Look-Up-Table-based Approach . . . 110

6.4 Potential Prime Numbers for Non LUT-Based Compression Functions . . . 111

6.4.1 Masked Compress Functions for Potential Prime Numbers . . . 111

7 CONCLUSION . . . 115

REFERENCES . . . 117

APPENDICES

A PYTHON IMPLEMENTATION . . . 123

A.1 Python Implementation of the Algorithm 13 . . . 123

A.2 Python Implementation of the Double and Check Algorithm as the Toy Example . . . 124

A.3 Python Implementation of the LUT Integration as the Toy Example . . . 125

A.3.1 Constructing the Look-Up-Table (LUT) . . . 126

A.3.2 The Look-Up-Table (LUT) . . . 126

A.3.3 Python Code of the Toy Example . . . 127

LIST OF TABLES

Table 4.1 Parameters of Kyber With Three Levels of Security 33

Table 6.1 Bit Representation, Index, and Values 107

Table 6.2 Analysis for the Prime Number 1153 112

Table 6.3 Analysis for the Prime Number 1409 113

Table 6.4 Analysis for the Prime Number 7681 114

LIST OF FIGURES

Figure 2.1 A Two-Dimensional Lattice With Two Different Basis Vectors (b_1, b_2). 8

Figure 2.2 Generating the $\mathbf{s}$ in the Kyber.CPAPKE.KeyGen() [13]. 19

Figure 4.1 Polynomial Multiplication [37] 49

Figure 4.2 Generating the $\mathbf{A}$ in the KeyGen() Algorithm [13]. 54

Figure 4.3 Generating the $\mathbf{s}$ in the KeyGen() [13]. 55

Figure 4.4 Generating the $\mathbf{e}$ in the KeyGen() [13]. 55

Figure 4.5 Generating the (pk, sk) in the KeyGen() [13]. 55

Figure 4.7 Bob Generating the $\mathbf{u}$ and v [13]. 61

Figure 4.6 How Does Bob Computes the polynomial $v \in \mathbb{R}_q$ [13]? 61

Figure 4.8 Decryption of Message m [13]. 64

Figure 4.9 Explanation of the Decryption Phase Done by Alice [13]. 64

Figure 4.10 Reading the pk from sk 71

Figure 4.11 Reading the $H(pk)$ from sk 71

Figure 4.12 Reading the z from sk 72

Figure 4.13 Fujisaki-Okamoto Transform for Kyber Key Encapsulation Mechanism (KEM) 73

Figure 4.14 $\overline{K}'$ as a Secret Key Dependent Value 73

Figure 5.1 Kyber.CCAKEM.Decaps() in Detail, Updated from [14]. 75

Figure 5.2 1-Bit Conversion [14] 77

Figure 5.3 Two Disjoint Interval to Compress a Polynomial Coefficient into a Single Bit [15] 78

Figure 5.4 Toy Example for Unmasked Compression Function of CRYSTALS-Kyber . . . 79

Figure 5.5 Shifting the Polynomial Coefficient with Certain Offset to Mask It [14] . . . 80

Figure 5.6 Saber ($q = 2^k$), Shifted Compress^s_q Function [14] . . . 80

Figure 5.7 Offset Between the MSB and $\frac{q}{2}$ [14] . . . 81

Figure 5.8 Bitslicing the Sensitive Coefficient x [14] . . . 85

Figure 5.9 When the Most Significant Bit Is Set to 1 [14] . . . 85

Figure 5.10 An example, where a coefficient x is $(110001110100)_2$. This coefficient $x \in [0, q-1]$ is directly compressed to 1 [14]. . . . 85

Figure 5.11 An Example Coefficient Compressed to 1 . . . 87

Figure 5.12 Compression: Mapping Each Coefficient to a Single Bit. [14] . . . 88

Figure 5.13 Gadgets in the Algorithm 13 in [14]. . . . 92

Figure 6.1 Shifting the Polynomial Coefficient with Certain Offset to Mask It [14] . . . 100

LIST OF ABBREVIATIONS

ApproxSVP$_\lambda$	The Approximate Shortest Vector Problem
CBD	Central Binomial Distribution
CCA	Chosen Ciphertext Attack
CPA	Chosen Plaintext Attack
D-LWE	Decision Learning With Error
Dec	Decryption
Decaps	Decapsulation
Enc	Encryption
Encaps	Encapsulation
GAPSVP$_\lambda$	The Gap Shortest Vector Problem
IND-CCA2	Indistinguishability Under Adaptive Chosen Ciphertext Attack
IND-CPA	Indistinguishability Under Chosen Plaintext Attack
KeyGen	Key Generation
KEM	Key Encapsulation Mechanism
LUT	Look Up Table
LWE	Learning With Error
MLWE	Module Learning With Error
Mod-SIVP	The Module Shortest Independent Vectors Problem
Mod-SVP	The Module Shortest Vector Problem
MSB	Most Significant Bit
NTT	Number Theoretic Transform
PKE	Public Key Encryption
PQC	Post Quantum Cryptography
PRF	Pseudo-Random Function
RLWE	Ring Learning With Error
SCA	Side-Channel Attack
SIS	The Shortest Integer Solution Problem
XOF	eXtendable-Output Function

CHAPTER 1

INTRODUCTION

In our current digital lives, we rely on public key cryptography, whose security is based on the complexity of specific mathematical problems. For example, RSA depends on the difficulty of *integer factorization*, while the foundation of elliptic curve cryptography (ECC) is the *discrete logarithm* problem. Unfortunately, both cryptosystems are susceptible to polynomial-time quantum computer attacks, as demonstrated in P.W. Shor's foundational studies like [45, 53].

For this reason, in 2016, the National Institute of Standards and Technology (NIST) initiated the Post-Quantum Cryptography Standardization (PQC) process [35]. At the conclusion of the third round, CRYSTALS-Kyber was selected for standardization as a public-key encryption and key-establishment algorithm. Additionally, three other digital signature algorithms, CRYSTALS-Dilithium, FALCON, and SPHINCS+, were chosen for standardization.

Of the algorithms chosen for standardization, only SPHINCS+ doesn't depend on the hardness of lattice-based problems. This suggests a preference by algorithm designers for utilizing the hardness of lattice-based problems. Although CRYSTALS-Kyber is resistant to quantum computer attacks due to its reliance on the *Module*-Learning-with-Error problem in module lattices [30], implementation security remains a concern, particularly regarding vulnerability to side-channel attacks.

Side-channel attacks (SCA), a concept first coined by [29], present an increasing threat to the security of emerging post-quantum cryptography algorithms like Kyber. These attacks exploit meta-data, often derived from tactics like *timing analysis* [55]

or *electromagnetic* analysis [59], aiming to recover the secret key during algorithm execution [14]. In fact, various implementations types of Kyber including software and hardware are already succesfully targeted by side-channel attacks [19, 28, 61].

Among the countermeasures against side-channel attacks, *masking* stands out the most prominently used technique, as highlighted by foundational studies such as [18] and [46]. However, while block ciphers like AES can be protected using only *Boolean* masking [24, 32], adapting masking technique to post-quantum cryptography algorithms often necessitates *transitions* between *arithmetic* and *Boolean* masking (**A2B**). This introduces a challenge, emphasizing the need for efficiently implementing casts between these two masking methods [4].

Building on this, the first masked implementation of a *Ring*-Learning With Error only masks the CPA decryption of the algorithm [51]. To achieve IND-CCA2 security, the Fujisaki-Okamoto transform [21] should be applied, as demonstrated by its use in Kyber [13]. This accentuates the importance of not only masking the Kyber.CPAPKE.Dec (See Alg. 6), but also the entire CCA scheme (See Alg. 11).

Further advancements in the domain of masked implementation of Ring-LWE saw refinements such as those in [38], which proposed a first-order masking of a KEM. This not only integrated the concepts of [51] but also introduced improvements like new decoding algorithms [4, 34].

Notably, as countermeasure against side-channel attacks, many popular schemes, including Kyber, adopted prime modulo q, which, as observed in [34] and [8], introduced significant performance overheads in comparison to the more efficient power-of-two moduli (2^k), necessitating *adaptations in many prior algorithms*. However, these masked implementations faced challenges in security, with identified vulnerabilities to side-channel attacks [11].

On a different note, while the Saber scheme used power-of-two modulo and demonstrated an efficient first-order masking scheme, it wasn't immune to attacks, as evidenced by a subsequent *deep learning power analysis* attacks [36]. Importantly, this attack didn't entirely negate the protective measures of [6], but rather exploited *higher-order* leakages, suggesting alternative approaches for increasing the resistance

against side-channel attacks. For this reason, cryptographic algorithms have been increasingly studied to be masked against *higher-order* side-channel leakage attacks [14].

At the epicenter of this effort is the CRYSTALS-Kyber Key Encapsulation Mechanism (KEM). Previous research has successfully masked multiple Kyber components, such as polynomial operations (most notably, the *NTT multiplication* [20]), *addition*, and *subtraction*, as seen in Alg. 5. Kyber.CPAPKE.Enc() and Kyber.CPAPKE.Dec() algorithms are also already masked such that they operate on each share of each polynomial coefficient separately and in parallel. Moreover, symmetric operations like the hash function G (referenced in Alg. 11) and PRF (seen in Alg. 5) have been refined using the Keccak masking method, a procedure detailed in [3]. For tasks centered on the central binomial distribution (CBD_η) in Alg. 5, the masking methodology from [52], later expounded in [14], was adopted.

In the evolving landscape of masking CRYSTALS-Kyber against higher-order side channel attacks, one component posed a distinct challenge and remained largely unaddressed: the *compression* function, as delineated in Line 4 of Alg. 6. The function, defined as:

$$\text{Encode}_1(\text{Compress}_q(v - \text{NTT}^{-1}(\hat{\mathbf{s}}^T \circ \text{NTT}(\mathbf{u})), 1)) \tag{1.1}$$

Masking this function is especially critical, given its direct processing of the secret key $\mathbf{s}$. With the vector of polynomials $\mathbf{u}$ and a polynomial v publicly known, adversaries could target this function in hopes of reverse-engineering to extract the secret vector of polynomials $\mathbf{s}$ making it a pivotal point of vulnerability.

Previous studies, working on power-of-module schemes like *Saber*, followed a *trivial approach* called the "Most Significant Bit" (MSB). However, when attempting this method to Kyber, challenges emerged due to Kyber's distinctive prime modulo $q = 3329$. The MSB method, tailored for algorithms with a modulo of the form $q = 2^k$, doesn't neatly fit Kyber's coefficient domain because Kyber's modulo isn't a power of two. To understand this better, Kyber's MSB is computed as $2^{\lceil log_2(q) \rceil - 1}$, which evaluates to $2^{\lceil log_2(3329) \rceil - 1} = 2^{\lceil 11.7 \rceil - 1} = 2^{12-1} = 2^{11} = 2048$. However, half of Kyber's modulo is $\lfloor q/2 \rfloor = \lfloor 3329/2 \rfloor = \lfloor 1664.5 \rfloor = 1664$. This difference results in an offset of $2048 - 1664 = 832$, causing the MSB method to be unsuitable for

CRYSTALS-Kyber. Because, in this case, there are 832 numbers which needs to be compressed to 1, but compressed to 0 (Refer to Fig. 5.7).

To address this, subsequent researches [17, 25] ventured into fully-masked implementations of Kyber, deriving their techniques from the compression function as introduced in [14]. Yet, even as the bit-sliced masking methods generally stood firm against side-channel attacks, revelations from recent investigations [60] spotlighted their vulnerabilities.

Building upon the research [14], we propose two alternative approaches for masking the compression function in Kyber.

First, we delve into the intricacies of the *Double and Check* method. At its core, this method is driven by motivation of adding another layer of obfuscation. Here's how it works: each coefficient of a polynomial is divided into "shares", which, when summed up in modulo q, equals the original value. Then, each share (let us denote the k-th share of a polynomial coefficient as $(a_i^{(k)_B})$ is shifted by a certain offset $(a_i^{(k)_B} \leftarrow a_i^{(k)_B} + \lfloor \frac{q}{4} \rfloor \mod q)$ to have a symmetry around the $\lfloor \frac{q}{2} \rfloor$ in the coefficient domain $([0, q-1])$. Next, the algorithm add each share to itself $(a_i^{(k)_B} + a_i^{(k)_B})$, to check whether

$$a_i^{(k)_B} + a_i^{(k)_B} \stackrel{?}{=} a_i^{(k)_B} + a_i^{(k)_B} \mod q.$$

If this is the case, we say that the share is less than $\lfloor \frac{q}{2} \rfloor$, and therefore compressed to 0. Otherwise, it is compressed to 1. The benefit of this algorithm is that the result of the masked compression function is derived from the outcome of an if-else statement, which does not depend on the secret vector of polynomial $\mathbf{s}$, and therefore does not require masking. In fact, since we do not directly double the share but instead add it to itself, it still yields a polynomial-time result, further protecting against time-analysis-based side-channel attacks.

Secondly, we delve into the time-honored memory/time trade-off through the *Look-Up-Table* (LUT) integration with 32 entities. The motivation here is rooted in the demands of cryptographic applications where computational speed is paramount. By adopting the LUT-based method, we have managed to expedite the compression process while retaining precision, thus catering to scenarios where rapid response times are essential. The integration of LUT showcases our commitment to striking a balance

between swift computations and optimal memory utilization, making it a quintessential asset for time-sensitive cryptographic operations.

In our study of the bit-wise compression function presented in [14], we questioned whether there exists a prime number q for which, when examining the most significant bit (MSB) of the binary representation of $\lfloor \frac{q}{2} \rfloor$, it would suffice to inspect only the 3rd, 4th, or 5th MSB bits. While we couldn't identify any prime numbers for the 3rd and 4th MSB bits, we did find a few for the 5th MSB bits. Among these, 7681 emerged as the best prime number for a new fast masked compression function. Additionally, it provides the smallest prime number satisfying $q \equiv 1 \mod 2n$ where $n = 256$, making it ideal for fast NTT multiplication. Notably, the prime number 7681 is also suggested by the main article on CCA2-secure CRYSTALS-Kyber [13], leading us to develop the new compression function.

In essence, our contribution lies in secret-value independent working mechanism of the *Double and Check* method with the efficiency of the LUT integration, offering a twofold approach that augments both security and performance in the realm of cryptographic compression.

The thesis is structured into seven chapters.

Chapter 2 offers essential background on lattice-based cryptography. This foundation is crucial for comprehending the *Module* Learning-With-Error problem upon which the security of CRYSTALS-Kyber rests. Chapter 3 delineates the definitions of IND-CPA and IND-CCA2 security. It also elucidates the intrinsic nature of *masking*, supplemented by illustrative examples that cover both *Boolean* and *arithmetic masking*. In Chapter 4, we detail the parameters and functions inherent to CCA2 secure CRYSTALS-Kyber, and present the three principal components (KeyGen, Enc, Dec) pertaining to both IND-CPA and IND-CCA2 secure Kyber. Chapter 5 delves into the unmasked compression function employed in Kyber, explores existing techniques to mask a compression function operating in 2^k modulo, why they cannot be applied to Kyber, and discusses the bit-sliced binary search method, as proposed by [14], to mask the compression function with any prime number. Chapter 6, the core contribution of this thesis, introduces the *Double-and-Check* algorithm (See Alg. 14), in addition to the *Look-Up-Table integration* for masking (See Alg. 15). We also

introduce how the prime number $q = 7681$ facilitates fast NTT multiplication and introduces a new fast masked compression function. Finally, Chapter 7 concludes the thesis, offering closing remarks on our proposed algorithms.

CHAPTER 2

INTRODUCTION TO LATTICE-BASED CRYPTOGRAPHY

In the forthcoming chapters, we will be using definitions related to lattice-based cryptography. To provide a comprehensive understanding, this section will briefly summarize these definitions. For those interested in gaining a more in-depth introduction to lattices and lattice-based cryptography, refer to the lecture notes by Chris Peikert [41] and Oded Regev [49], and following survey papers [33, 40, 44].

2.1 Introductory Definitions and Theorems for Lattice-based Cryptography

A brief definition of a lattice is given as follows.

Definition 2.1.1 (Lattice and Basis in Euclidean Space)**.** A *lattice* is a discrete set of points in space that can be generated by taking linear combinations of a set of basis vectors, where the coefficients are integers.

More formally, given a set of m linearly independent vectors $(\mathbf{b}_i)_{1\leq i\leq m}$ in $\mathbb{R}^n$, we can create a lattice $\mathcal{L} \subseteq \mathbb{R}^n$ by taking all possible linear combinations of these vectors with integer coefficients, i.e., by forming the following set:

$$L(\mathbf{b}_1, \ldots, \mathbf{b}_m) = \mathbf{x} \in \mathbb{R}^n : \mathbf{x} = \{\sum_{i=1}^{m} z_i \mathbf{b}_i, \ldots, z_i \in \mathbb{Z}\}$$

In this way, we obtain discrete subgroup of n*-dimensional Euclidean space*.

The set $(\mathbf{b}_i)_{1\leq i\leq m}$ is called *basis* of the lattice $\mathcal{L}$. We can represent this basis as a matrix B, where B is an $n \times m$ matrix whose columns are $\mathbf{b}_1, \mathbf{b}_2, ..., \mathbf{b}_m$. We can also represent the lattice $\mathcal{L}$ as $L(B) = L(\mathbf{b}_1, ..., \mathbf{b}_m)$. The rank of the lattice $\mathcal{L}$ is

equal to m, and its dimension is equal to n. When $n = m$, we refer to the lattice as a *full-rank* lattice.

It is important to note that a lattice can have many possible bases. However, any two bases $B_1, B_2 \in \mathbb{R}^{n \times m}$ will generate the same lattice if and only if $B_2 = B_1 \cdot U$, where U is an integer matrix whose determinant is either 1 or -1. This is known as a unimodular transformation.

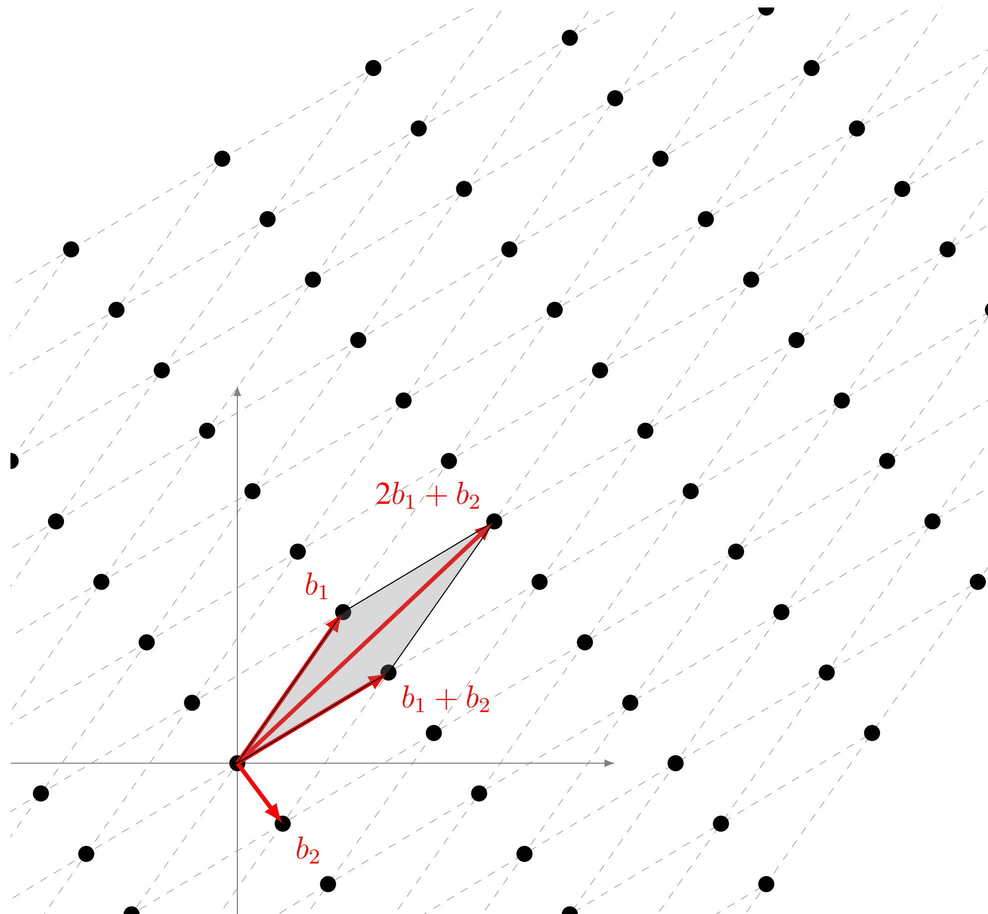


Figure 2.1: A Two-Dimensional Lattice With Two Different Basis Vectors (b_1, b_2).

Definition 2.1.2 (Lattice Determinant and Minimal Vector Norm)**.** Let $\mathcal{L}$ be a lattice and $(\mathbf{b_i})_{1 \leq i \leq m}$ be a basis of $\mathcal{L}$. The determinant of $\mathcal{L}$ is defined as the square root of the determinant of the matrix whose entries are the pairwise inner products of the basis vectors, i.e.,

$$\det(\mathcal{L}) = \sqrt{\det\left(\langle \mathbf{b_i}, \mathbf{b_j} \rangle\right)_{1 \leq i,j \leq m}}.$$

This definition is equivalent to the volume of the parallelepiped spanned by the basis vectors and is independent of the choice of basis.

For any $p \in \mathbb{N}^* \cup \{\infty\}$ and any lattice $\mathcal{L} \subset \mathbb{R}^n$, $\lambda_p(\mathcal{L})$ denotes the p-th power of the p-norm (ℓ_p) of the shortest non-zero vector in $\mathcal{L}$. In particular, when $p = 2$, which is the most commonly used norm, $\lambda_1(\mathcal{L})$ is simply the length of the shortest non-zero vector in $\mathcal{L}$ when measured in the usual way with a ruler, as we usually do with vectors in two or three dimensions.

Theorem 2.1.1 (Minkowski's First Theorem on Lattice Vector Lengths)**.** Let $\mathcal{L}$ be a full-rank lattice in $\mathbb{R}^n$ with determinant $\det(\mathcal{L})$. *Minkowski's First Theorem* states that $\mathcal{L}$ contains a non-zero vector $\mathbf{v}$ of length at most $\sqrt{n} \cdot |\det(\mathcal{L})|^{1/n}$. In other words, there exists a non-zero vector $\mathbf{v}$ in $\mathcal{L}$ such that

$$\lambda_1(\mathcal{L}) = ||\mathbf{v}||_2 \leq \sqrt{n} \cdot |\det(\mathcal{L})|^{1/n}.$$

The bound implies that for a given lattice, the shorter the shortest non-zero vector, the better the lattice. This quantity is a fundamental tool in the analysis and design of lattice-based algorithms, where the quality of the lattice directly affects the efficiency and security of the algorithms.

2.2 Lattice-based Hard Problems

Minkowski's bound, which is a theorem that provides an upper bound on the norm of the shortest nonzero vector in a lattice, is often not tight. Therefore, it is of interest to find ways to compute or approximate the norm of the shortest nonzero vector in a lattice. We can formalize this problem using two definitions:

Definition 2.2.1 (The Approximate Shortest Vector Problem (ApproxSVPγ))**.** Given a lattice $\mathcal{L} \subseteq \mathbb{R}^n$, the goal of the ApproxSVPγ problem is to find a non-zero vector $\mathbf{v} \in \mathcal{L}$ such that its Euclidean norm $||\mathbf{v}||$ is at most γ times the length of the shortest nonzero vector in $\mathcal{L}$, which is denoted as $\lambda_1(\mathcal{L})$.

Definition 2.2.2 (The Gap Shortest Vector Problem (GapSVPγ))**.** Given a lattice $\mathcal{L} \subseteq \mathbb{R}^n$ and a positive real number d, the GapSVPγ problem is to determine whether the inequality $\lambda(\mathcal{L}) \leq d$ or the inequality $\lambda(\mathcal{L}) > \gamma \cdot d$ holds, where γ is a positive real number greater than 1 and $\lambda(\mathcal{L})$ denotes the length of the shortest nonzero vector in $\mathcal{L}$.

In the 2002 study by Micciancio and Goldwasser, it was found that the GapSVPγ and ApproxSVPγ problems become equivalent when the parameter γ is set to 1 [22]. However, for γ greater than 1, the GapSVPγ problem gets easily reduced to the ApproxSVPγ problem [22]. It is possible to reduce the ApproxSVPγ problem to the GapSVPγ' problem with a randomized reduction that preserves the dimension of the lattice. The resulting GapSVPγ' problem has a parameter γ' that is polynomially related to γ and the dimension n of the lattice in the following manner, $\gamma' = \gamma'^{(\mathcal{O}(n/logn))}$[58].

2.3 Short Integer Solution (SIS) Hard Problem

The *Short Integer Solution* (SIS) problem is a foundational challenge in lattice-based cryptography. It revolves around finding a non-zero vector with a small Euclidean norm that, when multiplied by a given full-rank matrix, yields the zero vector modulo some integer q. This problem's complexity and structural intricacies provide the basis for numerous cryptographic constructions, acting as a *stepping stone* to understanding advanced problems like Learning With Error (LWE).

Definition 2.3.1 (The Short Integer Solution (SIS) Problem)**.** Given a matrix $\mathbf{A} \in \mathbb{Z}_q^{n\times m}$ and a positive integer $\beta > q$, the Short Integer Solution (SIS) problem is to find a non-zero vector $\mathbf{x} \in \mathbb{Z}^m$ such that $\mathbf{Ax} \equiv \mathbf{0} \pmod q$ and the norm $||\mathbf{x}||$ is less than β.

In the SIS problem as defined, the matrix **A** is usually assumed to be a "full-rank" matrix, which means that its columns are linearly independent. This assumption is necessary for the problem to be well-defined and have a unique solution. In other words, the columns of **A** are independent vectors that span a subspace of dimension m, which is the column space of **A**. This subspace is sometimes denoted as $\mathrm{Col}(\mathbf{A})$. The SIS problem asks to find a non-zero vector **x** that belongs to the nullspace of **A** (i.e., satisfies $\mathbf{Ax} \equiv \mathbf{0} \pmod q$) and has small Euclidean norm $||\mathbf{x}|| < \beta$ (e.g. $x \in 0, \pm 1'$), where the Euclidean norm of a vector $\mathbf{x} = (x_1, x_2, \ldots, x_m)$ is defined as $||\mathbf{x}|| = \sqrt{x_1^2 + x_2^2 + \cdots + x_m^2}$.

If we were to use the usual matrix notation to simplify the notation, we would have

the following mathematical representation, where A is a full-rank matrix ($n = m$).

$$\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix} \times \begin{bmatrix} x_{11} \\ x_{21} \\ \vdots \\ x_{m1} \end{bmatrix} \equiv 0 \in \mathbb{Z}_q^n$$

In the next section, we are going to talk about a complimentary problem called Learning With Error (LWE).

2.4 Learning With Error (LWE) Hard Problem

Introduced by Oded Regev in his seminal 2005 paper "On lattices, learning with errors, random linear codes, and cryptography" [48], the Learning with Errors (LWE) problem has since become a cornerstone in the field of lattice-based cryptography.

Definition 2.4.1 (Search Learning With Errors (Search-LWE) Problem)**.** Consider positive integers n, m, and q and distributions χ_s and χ_e over the finite field $\mathbb{Z}_q$. The search-LWE problem, given the tuple $(n, m, q, \chi_s, \chi_e)$, is defined as follows: Sample a vector s from χ_s^n, $s \xleftarrow{\$} \chi_s^n$, and for each i ranging from 1 to m, sample a vector a_i uniformly from $\mathbb{Z}_q^n$, $a_i \xleftarrow{\$} U(\mathbb{Z}_q^n)$, and a scalar e_i from χ_e, $e_i \xleftarrow{\$} \chi_e$. Then compute $b_i = a_i \cdot s + e_i \pmod q$. The task is to recover the secret vector s given the pairs (a_i, b_i) for $i = \{1, \ldots, m\}$.

Specifically, for a given algorithm A, we define the associated advantage [56].

$$\mathrm{Adv}^{\mathrm{lwe}}_{n,m,q,\chi_s,\chi_e}(A) = \Pr\Big[s \xleftarrow{\$} \chi_s^n; a \xleftarrow{\$} U(\mathbb{Z}_q^n); e \xleftarrow{\$} \chi_e; \\ b_i = \langle a_i, s\rangle + e_i \mod q : A((a_i, b_i)_{i=1}^m) = s \Big].$$

The search-LWE problem asks us to find a secret vector $s \in \mathbb{Z}_q^n$ with given inner products and noise. If we were to have no noise introduced into these equations or if we were to now these errors, the problem would be easily solved with the *Gaussian elimination*. However, the presence of the noise term makes this problem much

harder. To model the noise term in the search-LWE problem, we need to introduce the Learning With Errors (LWE) distribution.

Definition 2.4.2 (LWE Distribution)**.** Let $n \geq 1$ be a positive integer, and let q be a prime power such that $q = poly(n)$. The LWE distribution, with parameters (n, q, χ), is a probability distribution over $\mathbb{Z}_q^n \times \mathbb{Z}_q$, defined as follows:

Given an error distribution χ over $\mathbb{Z}_q$ (Gaussian, i.e., $\alpha \cdot q > \sqrt{n}$) with an *error rate* $\alpha \ll 1$, we generate a uniformly random secret key $s_i \in \mathbb{Z}_q^n$ and an error vector $e_i \in \mathbb{Z}_q^n$, the components of which are independently sampled from χ. The LWE distribution then outputs the pair (a_i, b_i), where $a_i \in \mathbb{Z}_q^n$ is a uniformly random vector, and $b_i = \langle a_i, s_i \rangle + e_i$ represents the inner product of a_i and s_i with the addition of the error term e_i.

Definition 2.4.3 (Decision Learning With Error (D-LWE) Problem)**.** Suppose that n and q are given as positive integers, and let χ_s and χ_e be defined as distributions over the integers, $\mathbb{Z}$. We choose s according to the error distribution χ_s^n, denoted as $s \stackrel{\$}{\leftarrow} \chi_s^n$. We further define two oracles:

- The oracle $O_{\chi_e,s}$: we select **a** uniformly at random from $\mathbb{Z}_q^n$, and e from χ_e, denoted as $\mathbf{a} \stackrel{\$}{\leftarrow} U(\mathbb{Z}_q^n)$ and $e \stackrel{\$}{\leftarrow} \chi_e$, respectively. The oracle then returns the pair $(\mathbf{a}, \langle \mathbf{a}, s \rangle + e \mod q)$.
- The oracle U: we select **a** uniformly at random from $\mathbb{Z}_q^n$, and u uniformly at random from $\mathbb{Z}_q$, denoted as $\mathbf{a} \stackrel{\$}{\leftarrow} U(\mathbb{Z}_q^n)$ and $u \stackrel{\$}{\leftarrow} U(\mathbb{Z}_q)$, respectively. The oracle then returns the pair $(\mathbf{a}, u)$.

The decision version of the LWE problem with parameters (n, q, χ_s, χ_e) involves **distinguishing** between outputs from the **oracle** $O_{\chi_e,s}$ and the **oracle** U. For a specific algorithm A, we define the advantage as follows [56]:

$$\mathrm{Adv}_{n,q,\chi_s,\chi_e}^{\mathrm{dlwe}}(A) = \left| \Pr(s \stackrel{\$}{\leftarrow} \mathbb{Z}_q^n : A^{O_{\chi_e,s}}() = 1) - \Pr(A^U() = 1) \right| .$$

The decision-LWE problem is considered hard if it is computationally infeasible to distinguish between the two cases with non-negligible advantage. Under certain conditions on the parameters, which have been studied by various researchers including

[39, 49], the search and decision variants of the LWE problem are equivalent. This means that the problem of finding a secret value given certain constraints is essentially the same as the problem of distinguishing between outputs from the **oracle** $O_{\chi_e,s}$ and the **oracle** U.

2.4.1 Matrix Representation of the Learning-With-Error (LWE) Problem

The Learning With Error problem can be succinctly represented in terms of matrix-vector multiplication, offering a clear visualization of its parameters and their interactions. Here, the multiplication of an $n \times n$ matrix $\mathbf{A}$ with the secret vector s is perturbed by a noise vector e, yielding the vector t. This matrix representation encapsulates the core challenge: to recover the secret vector s in the presence of the noise introduced by e.

$$\begin{pmatrix} a_{00} & a_{01} & \cdots & a_{0,n-1} \\ a_{10} & a_{11} & \cdots & a_{1,n-1} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n-1,0} & a_{n-1,1} & \cdots & a_{n-1,n-1} \end{pmatrix} \cdot \begin{pmatrix} s_0 \\ s_1 \\ \vdots \\ s_{n-1} \end{pmatrix} + \begin{pmatrix} e_0 \\ e_1 \\ \vdots \\ e_{n-1} \end{pmatrix} = \begin{pmatrix} t_0 \\ t_1 \\ \vdots \\ t_{n-1} \end{pmatrix}$$

In this context, we're dealing with an $n \times n$ matrix, denoted by $\mathbf{A}$, which consists of n^2 elements or *coefficients*. Additionally, we have a secret vector $s \in \mathbb{Z}_q^n$ and noise vector $e \in \mathbb{Z}_q^n$ with small coefficients. The result of multiplying the matrix $\mathbf{A}$ by vector s, and then adding the error vector e, is given by $t =< \mathbf{A} \cdot s > +e \in \mathbb{Z}_q^n$.

Given the matrix $\mathbf{A}$ is composed of n^2 elements, it is evident that it would require $\mathcal{O}(n^2)$ *storage space*. Moreover, to calculate the multiplication of the matrix $\mathbf{A}$ and the secret vector s, it would necessitate n^2 multiplication operations. This implies that *computational complexit*y is $\mathcal{O}(n^2)$. As the value of n increases, the required storage and number of multiplications become significant, potentially posing challenges for systems with limited resources, such as Internet of Things (IoT) devices.

To overcome this computational overhead, researchers come with different methods like Ring Learning with Error (Ring-LWE) problem.

2.5 The Ring Learning With Error Problem (Ring-LWE)

The Ring-Learning with Errors (Ring-LWE) problem was first introduced by Lyubashevsky, Peikert, and Regev in 2010 [31]. It is an extension of the Learning With Error problem, distinguished by its *replacement of the underlying vector space with a ring structure*. This pivotal alteration paves the way for enhanced computational efficiency, as we will see in the following sections. Rooted deeply in cryptography, Ring-LWE focuses on discerning a secret from noisy samples present within a ring.

Given that CRYSTALS-Kyber [13] operates within the polynomial ring defined over

$$\mathbb{Z}_{3329}[x]/\langle x^{256} + 1\rangle,$$

we begin by detailing its definition.

Definition 2.5.1 (Ring of Polynomials Over $\mathbb{Z}_q$)**.** $\mathbb{R}_q = \mathbb{Z}_q[x]/\langle x^n + 1\rangle$ defines a ring of polynomials over the finite field $\mathbb{Z}_q$, modulo the ideal generated by the polynomial x^n+1. Within $\mathbb{R}_q$, polynomials have a degree d such that $d < n$, and their coefficients are selected from a uniform distribution of elements in $\mathbb{Z}_q$. Polynomials in $\mathbb{R}_q$ are deemed equivalent if their difference can be expressed as a multiple of $x^n + 1$.

For example, if $q = 7$ and $n = 3$, then $\mathbb{R}_7 = \mathbb{Z}_7[x]/\langle x^3 + 1\rangle$ contains polynomials of the form $a(x) = a_0 + a_1x + a_2x^2 \in \mathbb{R}_q$, where a_0, a_1, a_2 are elements of $\mathbb{Z}_7$, and x^3 is identified with -1 in $\mathbb{R}_q$. This means that, in $\mathbb{R}_q$, we have $x^3 = -1$, so $x^3 + 1 = 0$, and any polynomial that is a multiple of $x^3 + 1$ is considered to be equivalent to the zero polynomial.

The choice of the modulus q and the polynomial $x^n + 1$ is important for ensuring that $\mathbb{R}_q$ is a ring with nice algebraic properties. Specifically, the condition $q \equiv 1 \mod 2n$ ensures that every element in $\mathbb{R}_q$ has a unique inverse modulo q, and the polynomial $x^n + 1$ is irreducible over $\mathbb{Z}_q$, which means that $\mathbb{R}_q$ is a field. These properties are important for the security and efficiency of cryptographic schemes based on Ring-LWE.

2.5.1 Search Ring Learning With Error Problem

The search *Ring* Learning With Errors (Ring-LWE) problem is a mathematical challenge central to many cryptographic systems, offering a blend of efficiency and strong security assumptions. In the search variant of this problem, the task is to determine a secret value when provided with purposely perturbed (noisy) samples, with the "noise" being integral to the security of the underlying system. This noise acts as an obfuscating element, making it computationally difficult to discern the secret, even when multiple samples are available.

Specifically, we consider a ring

$$\mathbb{R}_q = \mathbb{Z}_q[x]/\langle x^n + 1\rangle$$

of polynomials modulo the ideal generated by $x^n + 1$, where n is a power of two ($n = 2^k$) and q is a prime modulus satisfying

$$q \equiv 1 \ mod \ 2n.$$

The ring $\mathbb{R}_q$ is equipped with the canonical basis $\{1, x, x^2, \ldots, x^{n-1}\}$, and we identify x^n with -1. Suppose that χ_s and χ_e are distributions over the $\mathbb{R}_q$, where is s and e are sampled as following: $s \xleftarrow{\$} \chi_s$ and $e \xleftarrow{\$} \chi_e$. In addition, the matrix a is uniformly sampled from the distribution $\mathbb{R}_q$ as follows: $a \xleftarrow{\$} \mathcal{U}(\mathbb{R}_q)$.

Definition 2.5.2 (Search Ring Learning-With-Error Problem)**.** In the Ring-LWE problem, an adversary receives noisy samples of the form $(a, b = a \cdot s + e) \in \mathbb{R}_q \times \mathbb{R}_q$. Here, s is drawn from the distribution χ_s over $\mathbb{R}_q$, a is uniformly selected from $\mathbb{R}_q$, and the small noise vector e is sampled from the error distribution χ_e over $\mathbb{R}_q$. The primary challenge for the adversary is to deduce the secret s from these samples with a high likelihood of success. If we denote the advantage of the adversary as $\mathcal{A}$, then the search version of the Ring-LWE problem can be characterized accordingly [56].

$$\mathrm{Adv}^{\mathrm{rlwe}}_{n,q,\chi_s,\chi_e}(\mathcal{A}) = \Pr\left[s \xleftarrow{\$} \chi_s; a \xleftarrow{\$} \mathcal{U}(\mathbb{R}_q); e \xleftarrow{\$} \chi_e; b = a \cdot s + e : \mathcal{A}(a, b) = s\right]$$

2.5.2 Decision Ring Learning With Error Problem

In the realm of Learning With Error problems, alongside the well-known search-LWE, the *decision* Ring-LWE emerges as another significant variant. Research has demonstrated that the *search* Ring-LWE problem and the *decision* Ring-LWE problem are equivalent, both under average-case reduction and direct worst-case reduction [12, 42, 43].

$$(n/\alpha) - \text{approx worst case} \leq \text{search-LWE} \leq \text{decision-LWE}$$

Delving deeper into the problem structure, let's assume that we're provided with positive integers n and q. We also have χ_s and χ_e, which are distributions defined over $\mathbb{R}_q$. We select the secret s and noise e such that $s \xleftarrow{\$} \chi_s$, $e \xleftarrow{\$} \chi_e$.

Two distinct oracles can be established:

- Oracle $O_{\chi_e,\chi_s,e,s}$: This oracle selects the secret vector s from the distribution χ_s over $\mathbb{R}_q$ and noise vector e from χ_e over $\mathbb{R}_q$, and returns the pair $(a, b = a{\cdot}s{+}e)$.
- Oracle U: This one chooses a and u uniformly from $\mathbb{R}_q$, $a \leftarrow \mathcal{U}(\mathbb{R}_q)$ and gives back the pair (a, u).

Definition 2.5.3 (Decision Ring-LWE Problem)**.** The goal of the decision Ring-LWE problem, given (n, q, χ_s, χ_e), is to differentiate between the output from oracle $O_{\chi_e,s}$ and that of oracle U.

To be more specific, when considering algorithm $\mathcal{A}$, we compute its advantage as:

$$\text{Adv}^{\text{drlwe}}_{n,q,\chi_s,\chi_e}(\mathcal{A}) = \left| \Pr(s \xleftarrow{\$} R_q : A^{O_{\chi_e,s}}()) = 1) - \Pr(A^U() = 1) \right| .$$

2.5.3 Matrix Representation of the Ring Learning With Error Problem

The Ring Learning With Error (Ring-LWE) problem presents an intriguing twist on the classic LWE problem, incorporating unique matrix structures that offer both storage and computational efficiencies. The matrix in the Ring-LWE problem exhibits a

special structure, where each column is derived from a simple rotation and negation of its predecessor. This unique setup allows for the entire matrix to be represented merely by its first column, leading to significant savings in storage.

$$\begin{pmatrix} a_{0,0} & -a_{n-1,0} & -a_{n-2,0} & \cdots & -a_{1,0} \\ a_{1,0} & a_{0,0} & -a_{n-1,0} & \cdots & \\ a_{2,0} & a_{1,0} & a_{0,0} & \cdots & \\ a_{3,0} & a_{2,0} & a_{1,0} & \cdots & \ddots \\ \vdots & \vdots & & \vdots & a_{0,0} \\ a_{n-2,0} & & & & \\ a_{n-1,0} & a_{n-2,0} & a_{n-3,0} & \cdots & a_{0,0} \end{pmatrix} \cdot \begin{pmatrix} s_0 \\ s_1 \\ \\ \\ \\ \vdots \\ s_{n-1} \end{pmatrix} + \begin{pmatrix} e_0 \\ e_1 \\ \\ \\ \\ \vdots \\ e_{n-1} \end{pmatrix} = \begin{pmatrix} t_0 \\ t_1 \\ \\ \\ \\ \vdots \\ t_{n-1} \end{pmatrix}$$

The first column of the matrix a is chosen to begin with. The creation of subsequent columns involves a shift from the first column by one, followed by the application of a negative sign to the last coefficient, resulting in $-a_{n-1,0}$ as the first coefficient of the second column. This implies that given just the first column, one can reproduce the entire matrix. This significantly reduces the necessary storage to n, thus bringing down the **storage complexity** to $\mathcal{O}(n^2)$.

Moreover, the columns of the matrix aren't entirely independent of each other; rather, each is a shifted version of another. This interrelation offers avenues for reducing the computational burden of the multiplication operation between the matrix $\mathbf{a}$ and the vector s. The **computational complexity** of the Ring-LWE problem, therefore, is $\mathcal{O}(n \cdot \log n)$.

In [31], a hardness result was established for the search version of ring-LWE, which follows the same outline as that in [49]. The main issue in adapting the proof of for ideal lattices is the error distribution ([50], *p.9*), which is a non-spherical Gaussian distribution in ring-LWE. Unlike LWE, we cannot hit any fixed noise distribution by simply adding more noise because the error distribution has n parameters. Instead, we must assume that the ring-LWE oracle works for a whole range of noise parameters. However, for simplicity, we assume that the error is taken from a spherical Gaussian distribution, which is shown to be hard in [57] or can be derived from [31].

2.6 Module Learning with Errors Problem (Module-LWE)

Security of the CRYSTALS-Kyber is built upon the hardness of the *Module* Learning With Error (LWE) problem in module lattices [16]. This scheme can be adjusted based on a specified security parameter k, detailed in Table 4.1. In this section, we delve into what is meant by the term "Module-LWE".

The The Module-LWE problem generalizes the Ring-LWE problem. While LWE and Ring-LWE operate over vector spaces or single polynomial rings, respectively, Module-LWE is based on modules over polynomial rings. To clarify, in Ring-LWE, elements are individual polynomials with coefficients derived from the modulus of the constructed ring. However, in the Module-LWE structure, upon which Kyber relies, elements can be either individual polynomials or vectors of polynomials. When we have $\mathbb{R}_q^{k=1}$, it denotes a ring of polynomials where each polynomial's coefficients are drawn from $\mathbb{Z}_q$. In other words, the problem become Ring-LWE. Meanwhile, elements from $\mathbb{R}^{k=2}$ or $\mathbb{R}^{k=3}$ represent vectors of polynomials. From a computer science perspective, this can be viewed as an array comprising polynomials with coefficients from $\mathbb{Z}_q$. This shift to a *vectors of polynomials* structure means that we work with modules, not just individual polynomials.

The generation of these k modules can be observed in Lines 9-12 of Alg. 4.

---

for $i = 0$ to $k - 1$ **do**

  $\mathbf{s}[i] \leftarrow \text{CBD}\eta_1(\text{PRF}(\sigma, N))$ ▷ Sample $\mathbf{s} \in \mathbb{R}_q^k$ from $B\eta_1$

  $N \leftarrow N + 1$

end for

As you can see, the secret vector $\mathbf{s} \in \mathbb{R}_q^k$ is actually a vector of polynomials, meaning there are k polynomials in a single module. The visualization of an element from $\mathbb{R}_q^k$ is as the following.

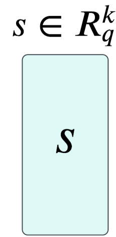


Figure 2.2: Generating the s in the Kyber.CPAPKE.KeyGen() [13].

If we consider $q = 5$ and $k = 3$ (for simplicity) with ideal polynomial $x^n + 1$, an example element from $\mathbb{R}_q^k$ would be a 3-tuple of polynomials where each coefficient is from $\mathbb{Z}_5$. Below, you will find a representation of this toy example.

$$\begin{pmatrix} 3x^2 + 2x + 1 \\ 4x^2 + x \\ 2x^2 + 3x + 4 \end{pmatrix}$$

In conclusion, $\mathbb{R}_q^k$ is not a ring but rather a **module** of rank k. In fact, $\mathbb{R}_q^k$ consists of k-tuples of elements from $\mathbb{R}_q$. Below, you will find the list indicating what the security parameter k corresponds to in CRYSTALS-Kyber:

- Kyber$512, k = 2$.
- Kyber$768, k = 3$.
- Kyber$1024, k = 4$.

2.6.1 The Hardness of the Module Learning with Errors

The Module-Learning with Error problem requires an adversary to distinguish between the real and uniformly selected pairs $(\mathbf{A}, \mathbf{b})$. The problem is considered as hard as there is no polynomial-time algorithm to solve with a non-negligible advantage.

$$
\begin{aligned}
(\mathbf{a}, b) &\leftarrow \mathbb{R}_q^k \times \mathbb{R}_q, \\
&\mathbf{a} \leftarrow \mathbb{R}_q^k, \quad \text{and} \quad b = \mathbf{a}^T\mathbf{s} + \mathbf{e}, \\
&\mathbf{s} \leftarrow \beta_\eta^k, \quad \text{and} \quad \mathbf{e} \leftarrow \beta_\eta^m.
\end{aligned}
$$

$$
\mathrm{Adv}_{\mathrm{mlwe}}(A) = \left| \begin{aligned} &\Pr\left[b' = 1 : \mathbf{A} \leftarrow \mathbb{R}_k^{m\times k}; (\mathbf{s}, \mathbf{e}) \leftarrow \beta_\eta^k \times \beta_\eta^m ; \right. \\ &\quad \left. \mathbf{b} = \mathbf{As} + \mathbf{e}, b' \leftarrow A(\mathbf{A}, b)\right] \\ &- \Pr\left[b' = 1 : \mathbf{A} \leftarrow \mathbb{R}_q^{m\times k}, b \leftarrow \mathbb{R}_q^m, b' \leftarrow A(\mathbf{A}, b)\right] \end{aligned} \right|
$$

To explain this more formally, the hardness of the Module-LWE problem is anchored in its underlying complexity, stemming from lattice problems, specifically the Module Shortest Independent Vectors Problem (Mod-SIVP) and the Module Shortest Vector Problem (Mod-SVP). The two presented theorems establish a formal connection between the hardness of these lattice problems and M-LWE.

In the work of Langlois and Stehlé (2005) [30, Theorem 4.7], lays out the quantum reduction from Mod-GIVP to M-LWE. The complexity is dependent on parameters such as the ring dimension (d), modulus (q), and the noise distribution (χ). When these conditions are satisfied, there exists a quantum algorithm that reduces the Mod-GIVP problem to an M-LWE problem, establishing its quantum hardness.

Langlois and Stehlé (2005) [30, Theorem 4.8] further extends this relationship, describing a classical polynomial-time reduction from the M-LWE problem defined over one modulus (q) to another M-LWE problem defined over a different modulus (p). Here, the transformation depends on the size of the modulus and the specific distribution of errors.

These reductions, under certain constraints on modulus size, ring dimension, and error distribution, demonstrate that as these parameters increase, the complexity and therefore the difficulty of solving the M-LWE problem also escalates. As such, M-LWE serves as a solid foundation for cryptographic schemes, owing to the conjectured hardness of these lattice problems

2.6.2 Matrix Representation of the Module-LWE

To understand the mathematical underpinnings of the Module-LWE problem, a matrix representation provides an insightful perspective. This section breaks down this representation and contrasts it with the familiar Ring-LWE setting.

$$\begin{pmatrix} A_{0,0}(X) & \cdots & A_{0,k-1}(X) \\ & \cdots & \\ \vdots & \vdots \; \ddots & \vdots \\ A_{k-1,0}(X) & \cdots & A_{k-1,k-1}(X) \end{pmatrix} \cdot \begin{pmatrix} s_0(X) \\ s_1(X) \\ \vdots \\ s_{k-1}(X) \end{pmatrix} + \begin{pmatrix} e_0(X) \\ e_1(X) \\ \vdots \\ e_{k-1}(X) \end{pmatrix} = \begin{pmatrix} t_0(X) \\ t_1(X) \\ \vdots \\ t_{k-1}(X) \end{pmatrix}$$

In the Module-LWE problem, the system employs a set of matrices A, each analogous to the structure used in the Ring-LWE setting. The key distinction is that **instead of one ring dimension**, the Module-LWE expands it to a module of rank k.

The structure of these matrices is particularly noteworthy. Despite the fact that we deal with k^2 matrices, we only need to store the first column of each matrix (n coefficients to be exact), thanks to their cyclical nature. This is exactly the case of Ring-LWE where the first row generates the whole ring matrix. Therefore, despite the added complexity of multiple dimensions in the module, the **storage complexity** becomes $\mathcal{O}(k^2 \cdot n)$, making it manageable and practical for larger dimensions.

Remember how the **computational complexity** of the Ring-LWE was $\mathcal{O}(n \cdot logn)$. Now, we have not 1, but k^2 matrices. Consequently the complexity of the Module-LWE becomes $\mathcal{O}(k^2 \cdot nlogn)$.

CHAPTER 3

TECHNICAL BACKGROUND

3.1 Indistinguishability Under Chosen Plaintext Attacks (IND-CPA)

In the field of cryptography, the security of a public key encryption scheme is a fundamental consideration. A public key encryption scheme, denoted by PKE $=$ $(\textbf{KeyGen}(), \textbf{Enc}(), \textbf{Dec}())$, is a "*triple of probabilistic algorithms together with a message space* $\mathcal{M}$" [13]. In this standard notion of the PKE, the **KeyGen**() algorithm outputs a public, pk, and a secret sk key pair:

$$(pk, sk) \leftarrow \textbf{KeyGen}().$$

The encryption algorithm **Enc()** takes the public key *pk* and a message $m \in \mathbb{M}$ as inputs to output the ciphertext, c. Then, we can say that a PKE scheme is $(1 - \delta) - correct$ [26] if

$$\textbf{E}[max_{m \in \mathbb{M}} \text{Pr}[\text{Dec}(sk, \text{Enc}(pk, m)) = m] \geq 1 - \delta,$$

where **E** denotes the expectation over randomly chosen public and secret keys generated by the **KeyGen**() algorithm.

The $IND - CPA^{\mathbb{A}}_{PKE}$ game is a well-known game used to formally define the security of PKE in terms of indistinguishability [9, 54]. In this game, a probabilistic polynomial-time adversary $\mathbb{A}$ aims to win the game in two stages. In the first stage, $\mathbb{A}$ is given access to an encryption oracle **Enc**() to encrypt a polynomially bounded number of messages of its choice. In the second stage, $\mathbb{A}$ submits two distinct fresh plaintext messages m_0 and m_1 from the message space $\mathcal{M}$ such that $|m_0| = |m_1|$, and

receives the ciphertext of one of the messages, c_b, where $b \leftarrow \{0, 1\}$. The adversary's goal is to decide which message m_b is encrypted in the given ciphertext.

$$
\begin{array}{l}
\underline{\textbf{Game } IND - CPA^{\mathbb{A}}_{PKE}\text{:}} \\
(pk, sk) \leftarrow KeyGen() \\
b \leftarrow \{0, 1\} \\
(m_0, m_1) \leftarrow \mathbb{A}^{\text{Dec}(\cdot)}(pk) \\
c_b \leftarrow Enc(pk, m_b) \\
b' \leftarrow \mathbb{A}^{\text{Dec}(\cdot)}(pk, c_b) \\
\textbf{return } b \stackrel{?}{=} b'
\end{array}
$$

For a PKE scheme to be IND-CPA-secure means that for all efficient adversaries $\mathbb{A}$, there exists some negligible function $negl(n)$ of the security parameter n such that the advantage of A in winning the $IND-CPA^{\mathbb{A}}_{PKE}$ game is negligible, i.e., significantly less than $\frac{1}{2}$. Formally, the advantage of A in winning the game is defined as follows.

$$
\textbf{Adv}^{IND-CPA}_{PKE}(\mathbb{A}) = \left| \Pr[IND - CPA^{\mathbb{A}}_{PKE} = 1] - \frac{1}{2} \right| < \textbf{negl}(n).
$$

3.2 Indistinguishability Under (Adaptive) Chosen Ciphertext Attacks (IND-CCA)

Indistinguishability under adaptive chosen ciphertext attacks is a primary concern in ensuring the security of Key Encapsulation Mechanisms (KEM) [47].

This means that an adversary is given access to an encapsulation oracle **Encaps()** during the attack, which allows them to encapsulate any number of keys of their choice. Additionally, the attacker has access to a decapsulation oracle, **Decaps()**. Compared to the standard security notion of indistinguishability under chosen plaintext attacks (IND-CPA), IND-CCA security provides even stronger security guarantees.

The IND-CCA game is used to formally define this security notion. In this game, a probabilistic polynomial-time adversary attempts to win the game in two stages. First,

the adversary generates a public and secret key pair using a key generation algorithm **KeyGen()**. In the second stage, the adversary submits two distinct bits (b and b') and receives an encapsulation of one of the keys, (c', K_0'). The adversary then attempts to determine which bit was used to encapsulate the key by submitting $(pk, c', K_{b'})$ to the decapsulation oracle **Decaps**$()$.

Game $\underline{IND - CCA^{\mathbb{A}}_{KEM}}$:

$(pk, sk) \leftarrow KeyGen()$

$b \leftarrow \{0, 1\}$

$K_1' \leftarrow K$

$(c', K_0') \leftarrow Encaps(pk)$

$b' \leftarrow \mathbb{A}^{Decaps()}(pk, c', K_{b'})$

return $b \stackrel{?}{=} b'$

By demonstrating that an efficient adversary's probability of winning the IND-CCA game is negligible, a KEM can be considered IND-CCA-secure. Specifically, given some negligible function **negl**(n) of the security parameter n, the advantage of an adversary in winning the game is defined as:

$$\mathbf{Adv}_{IND-CCA}(\mathbb{A}) = \left| \Pr[IND - CCA^{\mathbb{A}}_{KEM} = 1] - \frac{1}{2} \right| < \mathbf{negl}(n)$$

3.3 Masking: Dividing a Sensitive Value into Uniformly Selected Shares

Masking serves as an essential and intuitive countermeasure to safeguard lattice-based post-quantum cryptography (PQC) implementations against side-channel attacks.

3.3.1 Need for Masking

Let's assume we have a sensitive value $x \in \mathbb{Z}_q$ that *needs to be masked*. Fundamentally, masking separates this sensitive value x into multiple shares (suppose it is divided into n_s shares) to make sure that the underlying **circuit** (the algorithm itself) will process these shares in a secure way.

3.3.2 Circuits and Gadgets They Are Working on

For the first algorithm we will examine (see Algorithm 13 by [14]), the sensitive values that require masked processing are the coefficients of a polynomial $a \in \mathbb{Z}_q[X]$.

Each coefficient of a secret polynomial is split into uniformly sampled shares and **processed securely through a sequence of operations within the circuit**. The algorithm is transformed into a circuit by representing each operation with a corresponding set of interconnected logical gates and components. **Gadgets**, **which are specialized sub-circuits**, perform these operations on the shares of sensitive values.

The figure 5.13 displays the underlying circuit for the Masked Compress$_q(x, 1)$ Algorithm 13. The algorithm takes a sensitive coefficient x (in this case, each 256 coefficient, as $n = 256$, See 4.1) of a polynomial a that needs to be compressed into a bit-string of length 256 in a masked manner), splits it into shares, and conducts a series of operations on them using gadgets. These gadgets represent distinct operations that process the shares securely, ultimately producing the following circuit. This circuit ensures that the sensitive values, such as the coefficients of polynomials, are protected from side-channel attacks throughout the computation process.

The resulting tuple (n_s-tuple: as x is split into n_s shares, we call it a *tuple* now) is denoted as the following [14]:

$$n_s - \text{tuple} : x^{(\cdot)}$$

3.3.3 Arithmetic Encoding of a Sensitive Variable $x \in \mathbb{Z}_q$

This encoding represents a variable $x \in \mathbb{Z}_q$ as a set of n_s arithmetic shares, denoted by $x^{(\cdot)_A}$. Each individual share is represented as $x^{(i)_A}$, where $0 \leq i < n_s$, and is an element of the ring $\mathbb{Z}_q$ [14].

The key property of this arithmetic encoding is that the sum of all shares modulo q reconstructs the original variable,

$$x \equiv x^{(0)_A} + x^{(1)_A} + \cdots + x^{(n_s-1)_A} \mod q$$

This encoding scheme provides a secure means to process sensitive information by

breaking the variable into multiple shares. Each share only reveals information about itself. For example, if the sensitive variable x is divided into n_s shares, the algorithm exposes information related only to its specific share, such as x_1, x_2, and so forth. This makes it difficult for an adversary to extract meaningful data. Moreover, because the shares are uniformly selected from $\mathbb{Z}_q$ and a new secret key and noise vectors are generated each time, repeatedly running the algorithm does not offer any advantage to the adversary, as the arithmetic encoding varies with each iteration.

3.3.3.1 A Toy Example for an Arithmetic Encoding

Let's assume we have a sensitive variable $x \in \mathbb{Z}_q$ with $x = 42$. We will use a prime $q = 7$ as the modulus for our arithmetic encoding. We will also choose the number of shares, n_s, to be 3. The goal is to represent x as a set of n_s arithmetic shares such that the sum of all shares modulo q reconstructs the original variable. To create the shares, we can use the following process:

- Uniformly select two shares from $\mathbb{Z}_q$, say $x^{(0)_A}$ and $x^{(1)_A}$.
- Compute the third share, $x^{(2)_A}$, by subtracting the sum of the other shares from x modulo p.

For example, let's randomly choose the first two shares as $x^{(0)_A} = 2 \in \mathbb{Z}_7$ and $x^{(1)_A} = 5 \in \mathbb{Z}_7$. To compute the third share, we can use the following equation:

$$x^{(2)_A} \equiv x - x^{(1)_A} - x^{(0)_A} \mod q$$

Plugging in the values, we get:

$$\begin{aligned} x^{(2)_A} &\equiv 42 - 2 - 5 \mod 7 \\ &\equiv 35 \mod 7 \\ &\equiv 0 \end{aligned}$$

So, the three shares for $x = 42$ are $x^{(0)_A} = 2$, $x^{(1)_A} = 5$, and $x^{(2)_A} = 0$. Now, let's check if the sum of these shares modulo $q = 7$ reconstructs the original variable:

$$x \equiv x^{(0)_A} + x^{(1)_A} + x^{(2)_A} \mod 7$$
$$42 \equiv 2 + 5 + 0 \mod 7$$
$$42 \equiv 7 \mod 7$$
$$42 \equiv 0 \mod 7$$

3.3.4 Boolean Encoding of a Sensitive Variable $x \in \mathbb{Z}_2^k$

Building on foundational studies such as [18] and [46], which highlight the significance of masking in countering side-channel attacks, it's evident that masking Post-Quantum Cryptography (PQC) algorithms like Kyber necessitates not just one form of masking like arithmetic encoding. Instead, a blend of both Boolean and arithmetic masking is required. Consequently, we will introduce the concept of Boolean masking.

In the *Boolean masking* case,

$$a = a^{(0)_B} \bigoplus a^{(1)_B} \bigoplus \cdots \bigoplus a^{(n_s-1)_B} \mod \mathbb{Z}_2^k$$

the $\mathbf{n}_s$-tuple representation of a secret coefficient $a \in \mathbb{Z}_2^k$ can be denoted as $a^{(\cdot)_B} = (a^{(0)_B}, \cdots a^{(n_s-1)_B})$, which comprises n_s Boolean shares $a^{(i)_B}$ with $0 \leq i < n_s$.

3.3.4.1 A Toy Example for Boolean Encoding of a Sensitive Variable $x \in \mathbb{Z}_{2^k}$

Consider a sensitive variable $x \in \mathbb{Z}_{2^k}$, where $x = 26$. Define $k = 5$ to represent values ranging from 0_{10} to 31_{10}, or in binary from 00000_2 to 11111_2. Therefore, $x = 11010_2$. Let the number of shares, n_s, be 3. The aim is to represent x as n_s Boolean shares so that the bitwise XOR of all the shares reconstructs the original variable x.

To generate the shares, we can adopt the following procedure:

- Randomly select two binary shares from the space $\mathbb{Z}_{2^k}$, denote these as $x^{(0)_B}$ and $x^{(1)_B}$.

- Calculate the third share, $x^{(2)_B}$, by performing a bitwise XOR operation on x and the two previously generated shares.

Suppose the first two shares are randomly selected as $x^{(0)_B} = 10101_2$ and $x^{(1)_B} = 00100_2$. The third share is computed using the bitwise XOR equation:

$$\begin{aligned} x^{(2)_B} &= x \oplus x^{(0)_B} \oplus x^{(1)_B} \\ x^{(2)_B} &= 11010_2 \oplus 10101_2 \oplus 00100_2 \\ x^{(2)_B} &= 01011_2 \end{aligned}$$

Hence, the three shares for $x = 26$ are $x^{(0)_B} = 10101_2$, $x^{(1)_B} = 00100_2$, and $x^{(2)_B} = 01011_2$. Validating the correctness of these shares, we can verify:

$$\begin{aligned} x &= x^{(0)_B} \oplus x^{(1)_B} \oplus x^{(2)_B} \\ 11010_2 &= 10101_2 \oplus 00100_2 \oplus 01011_2 \\ 11010_2 &= 11010_2 \end{aligned}$$

3.3.5 Security Definitions for Shared Implementations

To explain the security of these shared implementations in a more formal way, researchers Ishai, Sahai, and Wagner developed the *t-probing model*, which they introduced in their paper [27].

The t-probing model in cryptography represents an adversary with access to up to t intermediate variables in the system. This model is essential for evaluating the security of masked circuits against side-channel attacks, where the goal is to ensure that any combination of t variables within the circuit does not reveal information about the secret.

The t-NI and t-SNI security notion was first introduced in [2] and [3], and the notions used in the article [14] were derived from [10].

3.3.5.1 t-NI Security Notion

Suppose we have a single gadget G (See the 13 gadgets labeled in the comments given in the Algorithm 13, and to the Figure 5.13 for an example).

This gadget G receives $x^{(\cdot)}$ ($n_s - tuple$) as its input and produces $y^{(\cdot)}$ as its output.

Here, $x^{(\cdot)}$ represents an (n_s)-tuple, which means it is a sequence or an ordered list of n_s elements. For instance, if we consider a simple case where $n_s = 3$, an example of such (3)-tuple would be: $x^{(\cdot)} = (x^{(1)}, x^{(2)}, x^{(3)})$. The same case applies to the $y^{(\cdot)}$.

We say that G is t-NI (Non-Interference ([2, 3]) secure if it pertains to the following properties:

For any collection of t_G intermediate variables (where $t_G \leq t$), there exists a subset $\mathcal{I}$ contained within the index range $[0, n_s - 1]$ of input indices. This subset $\mathcal{I}$ must satisfy the condition that its size is less than or equal to t_G ($|\mathcal{I}| \leq t_G$).

So far we have

$$t_G \leq t$$
$$|\mathcal{I}| \leq t_G \text{ where } \mathcal{I} \subset [0, n_s - 1].$$

The crucial aspect is that t_G can be *perfectly simulated* [14] from the $x^{(\mathcal{I})}$ input indices.

In simpler terms, this means that even if an adversary gets access to up to $t_G \leq t$ intermediate variables in the gadget, they can't gain any more information than what is already available from a subset of the input shares. This is crucial for maintaining the security of the gadget as it prevents sensitive information from being leaked through side-channel attacks.

3.3.5.2 t-SNI Security Notion

Given a gadget G, which receives $x^{(\cdot)}$ as its input and produces $y^{(\cdot)}$ as its output, the t-SNI (Strong Non-Interference ([3])) security level refers to the following properties:

For any set of t_G intermediate variables (where $t_G \leq t$) and any subset O of output indices within the index range [0, n_s - 1], provided that $t_G + |O| \leq t$, there exists a subset $\mathcal{I}$ contained within the index range $[0, n_s - 1]$ of input indices. The size of this subset $|\mathcal{I}|$ should be less than or equal to t_G.

Crucially, given only the input shares $x^{(I)}$ (i.e., the inputs corresponding to the indices in the subset I), it is possible to perfectly simulate the values of these t_G intermediate variables and the output variables $y^{(O)}$.

In simpler terms, the gadget G is t-SNI secure if, even if an attacker gets access to up to t values, which may include any t_G intermediate variables and any output shares $y^{(O)}$, they can't gain any more information than what is already available from a subset of the input shares.

CHAPTER 4

CRYSYALS-KYBER KEY ENCAPSULATION MECHANISM (KEM)

In this section, we will explore CRYSTALS-Kyber, setting aside discussions of masked implementation for now.

4.1 Parameters to be Used

CYRSTALS-Kyber is a Post-Quantum Cryptography (PQC) key encapsulation mechanism (KEM) that employs a prime modulus, denoted by

$$q = 3329$$

in its operations. In contrast, Saber utilizes a power-of-two modulus (for example, 2^k). This design choice introduces a significant computational overhead in masked implementations compared to a 2^k modulus, as evidenced by the study of [34]. Table 4.1 below lists the corresponding parameters for given security levels k. In this section, we will discuss each of these parameters and their roles within the Kyber.CPAPKE.Enc() algorithm (See Alg. 5).

Table 4.1: Parameters of Kyber With Three Levels of Security

	n	k	q	η_1	η_2	(d_u, d_v)	δ
KYBER512	256	2	3329	3	2	(10,4)	2^{-139}
KYBER768	256	3	3329	2	2	(10,4)	2^{-164}
KYBER1024	256	4	3329	2	2	(11,5)	2^{-174}

- The δ denotes the failure probability of decryption [1]. Note that the δ prob-

ability can be computed by an publicly available Python script, which can be accessed from here [1].

- The value of n is set to 256, implying that Kyber operates with polynomials $a(x)$ from $\mathbb{Z}[X]_{3329}$ consisting of 256 coefficients, with each coefficient drawn from the finite field $\mathbb{Z}_{3329}$. The key aspect of the Kyber cryptosystem is efficient polynomial multiplication which is achieved by using the Number Theoretic Transform (NTT). The NTT requires the modulus, represented by the parameter q, to be a specific kind of prime number such that n divides $q-1$. The choice of $q = 3329$ satisfies this condition with $n = 256$. There are indeed two smaller primes, 257 and 769, which satisfy this criterion, but they would lead to a non-negligible failure probability. This would be unacceptable as it could compromise the CCA2 security level that Kyber aims to maintain.

- The parameter η_1 dictates the generation of the secret vector s in $\mathbb{R}_q^k$ and the noise vector e in $\mathbb{R}_q^k$, as seen in Kyber.CPAPKE.KeyGen() (refer to Lines 10 and 14 in Alg. 4, respectively). It also governs the noise in $\mathbf{r}$ in $\mathbb{R}_q^k$ in the Kyber.CPAPKE.Enc() (see Line 5 in Alg. 5). It should be noted that the vectors s and e in CRYSTAL-Kyber are sampled from a *Centered Binomial Distribution* (CBD) (refer to Alg. 2). Later in this section, we will provide a comprehensive explanation of this. Vectors are denoted in bold.

$$\mathbf{s} \xleftarrow{\$} B_{\eta_1}$$
$$\mathbf{e} \xleftarrow{\$} B_{\eta_1}$$
$$\mathbf{r} \xleftarrow{\$} B_{\eta_1}$$

- The η_2 manages the noise in variables $\mathbf{e}_1 \in \mathbb{R}_q^k$ and $e_2 \in \mathbb{R}_q$.

$$\mathbf{e}_1 \xleftarrow{\$} B_{\eta_2}$$
$$e_2 \xleftarrow{\$} B_{\eta_2}$$

- (d_u, d_v) are related to the compression and decompression functions, denoting in how many bits that the message/polynomial will be compressed.

[1] https://github.com/pq-crystals/security-estimates.

4.2 Notations and Background Knowledge for the CPAKEM and CCAKEM Secure CRYSTAL-Kyber Algorithms

In this section, we will provide the necessary notation that is used in the pseudo-code of the algorithms given below.

- Kyber.CPAKEM.KeyGen() - Refer to Algorithm 4
- Kyber.CPAPKE.Enc(pk, m, r) - Refer to Algorithm 5
- Kyber.CPAPKE.Dec(c, sk) - Refer to Algorithm 6
- Kyber.CCAKEM.KeyGen() - Refer to Algorithm 7
- Kyber.CCAKEM.Enc(pk) - Refer to Algorithm 9, and
- Kyber.CCAKEM.Dec(c, sk) - Refer to Algorithm 11

4.2.1 Byte and Byte Arrays

The set $\mathcal{B}$ is defined as $\{0, \ldots, 255\}$, representing 8-bit integers. An element in this set, termed a **byte**, consists of 8 binary digits (bits). As these bytes are unsigned, they solely denote non-negative integers.

$\mathcal{B}^k$ denotes the set of all byte arrays of length k. Essentially, a byte array is a sequence containing k bytes. For instance, in Alg. 4, we see that output of Kyber.CPAPKE.KeyGen() is secret key such that $sk \in \mathcal{B}^{12 \cdot k \cdot n/8}$. If we were to examine that in great detail, we would see that **s** is actually a vector of k polynomials ($\mathbf{s} \in \mathbb{R}_q^k$) sampled from B_{η_1}, each with n coefficients; given that a coefficient from $\mod 3329$ is 12-bit long, then we multiply by $k \times n \times 12$ and divide by 8 to get the byte length.

In other example, if k were to be 3, $\mathcal{B}^3$ refers to the set of all possible byte arrays that have a length of 3 bytes. Here are a few examples of byte arrays that belong to $\mathcal{B}^3$:

- $[0, 0, 0]$: This array consists of three bytes, all set to the minimum value of 0.
- $[255, 255, 255]$: This array consists of three bytes, all set to the maximum value of 255.

- $[0, 127, 255]$: This array consists of three bytes with different values (0, 127, and 255).

Each byte in the array can have a value from 0 to 255, as explained earlier. In other words, each element of the array is a byte (an element of B). So, there are $256^3 = 16,777,216$ unique byte arrays with a length of 3 bytes in the set $\mathcal{B}^k$.

The expression $m \in \mathcal{B}^{32}$ indicates that the message m is uniformly drawn from the set $\{0, 1\}^{256}$. See how Alg. 5 takes input message $m \in \mathcal{B}^{32}$ to encypt it. Given that each byte has 8 bits, the total bit-length of the message m is $32 \times 8 = 256$ bits.

$$m \in \mathcal{B}^{32} = m \leftarrow \{0, 1\}^{256}$$

$\mathcal{B}^*$ denotes the set of byte arrays of arbitrary length, also known as byte streams. This set includes byte arrays of any length, from empty arrays to arrays with an infinite number of elements.

The $(a||b)$ notation represents the concatenation of two byte arrays a and b. The resulting byte array has a length equal to the sum of the lengths of a and b. For instance, the output of Kyber.CPAPKE.Enc() algorithm is the two public ciphertext $c \leftarrow (c_1 \mid\mid c_2)$ (Refer to Line 23 in Alg. 5).

When working with byte arrays, you may need to extract a portion of the array or refer to a specific byte within the array. In this context, the notation "$a + k$" is used to represent the byte array that starts at the k-th byte of the original byte array "a" (with indexing starting at zero).

- Let "a" be a byte array of length l.
- Let "b" be another byte array.
- Let "c" be the concatenation of "a" and "b", denoted as $(a||b)$.

Now, if we look at the concatenated byte array "c", the first part of "c" consists of the bytes from "a". The second part of "c" consists of the bytes from "b". Since "a" has a length of l, the byte array "b" starts at the $(l + 1)$-th position in "c" (remember that

indexing starts at zero).

In this case, "$a + l$" refers to the byte array starting at the l-th byte of "a". Since "b" starts at the l-th byte of the concatenated byte array "c", we can say that "$b = a + l$" in this particular example.

In summary, the notation "$a + k$" is used to denote a byte array that starts at the k-th byte of the original byte array "a". In the example provided, "b" is found in the concatenated byte array "c" starting at the l-th byte, so "$b = a + l$".

The *Byte to Bits Function* function converts a byte array of length l into a bit array of length $8l$. Each byte in the input array is converted into 8 bits in the output array. The function computes each bit β_i at position i of the output bit array from the corresponding byte $b_{i/8}$ at position $i/8$ of the input array using the formula:

$$\beta_i = (b_{i/8}/2^{(i \mod 8)}) \mod 2 \tag{4.1}$$

Now let's go through an example.

- Input byte array (in hexadecimal): A5
- Input byte array (in binary): 10100101

We have $l = 1$ (one byte), so the output bit array will have $8l = 8$ bits. We'll use (4.1) for i ranging from 0 to 7.

- $i = 0$
  - $i/8 = 0/8 = 0$
  - $i \mod 8 = 0$
  - $2^{(i \mod 8)} = 2^0 = 1$
  - $b_{i/8} = b_0 = A5$ (in decimal: 165)
  - $\beta_0 = (165/1) \mod 2 = 1$

- $i = 1$
  - $i/8 = 1/8 = 0$ - *integer division*
  - $i \mod 8 = 1$
  - $2^{(i \mod 8)} = 2^1 = 2$
  - $b_{i/8} = b_0 = A5$ (in decimal: 165)
  - $\beta_1 = (165/2) \mod 2 = 1$

Continuing the process for $i = 2$ to 7, we get the output bit array $\beta = [1, 0, 1, 0, 0, 1, 0, 1]$. This matches the binary representation of the input byte array A5 to $(10100101)_2$.

4.2.2 The Polynomial Ring $\mathbb{R}_q$ Represented as $\mathbb{Z}_q[X]/(X^n + 1)$

$\mathbb{R}$ is the quotient ring $\mathbb{Z}[X]/(X^n + 1)$. Here, $\mathbb{R}$ is a polynomial ring wherein its elements are polynomials with coefficients in $\mathbb{Z}$. This ring is formed by taking the quotient of the polynomial ring $\mathbb{Z}[X]$ by the ideal generated by $X^n + 1$. Any polynomial in the ring $\mathbb{R}$ can be reduced modulo $X^n + 1$, and any two polynomials that differ by a multiple of $X^n + 1$ are considered equivalent in this ring. So, the polynomials in $\mathbb{R}$ are of the form

$$f(X) = a_0 + a_1X + ... + a_{(n-1)}X^{(n-1)},$$

where a_i are integers, and $f(X)$ is considered equivalent to $g(X)$ if their difference $(f(X) - g(X))$ is divisible by $X^n + 1$.

$\mathbb{R}_q$, on the other hand, is the ring $\mathbb{Z}_q[X]/(X^n + 1)$. $\mathbb{R}_q$ is also a **polynomial ring**, but its **elements have coefficients in the integers modulo** q $(\mathbb{Z}_q)$. Similar to $\mathbb{R}$, the ring $\mathbb{R}_q$ is constructed by taking the polynomials in $\mathbb{Z}_q[X]$ and imposing the relation $X^n + 1 = 0$. Polynomials in $\mathbb{R}_q$ have the same form as in $\mathbb{R}$, but the coefficients a_i are integers modulo q.

Let's take a simple example with a smaller value of n and q to illustrate the concept.

Suppose $n = 2$ and $q = 5$. Then, we have the ring $\mathbb{R}_5 = \mathbb{Z}_5[X]/(X^2 + 1)$. In this ring, the relation $X^2 + 1 = 0$ is imposed. So, any polynomial that has a term with a

power of X^2 or higher can be reduced modulo $X^2 + 1$. For example, let's consider the polynomial $f(X) = 3X^2 + 2X + 4$ in $\mathbb{Z}_5[X]$. In the ring R_q, we can reduce $f(X)$ modulo $X^2 + 1$ as follows:

$$\begin{aligned} f(X) &= 3X^2 + 2X + 4 \\ &\equiv -2X + 2X + 4 \ (mod \ X^2 + 1) \\ &\equiv 3X + 4 \ (mod \ X^2 + 1) \end{aligned}$$

So, in the ring $\mathbb{R}_q$, the polynomial $f(X)$ is equivalent to the polynomial $3X + 4$, and any operation involving $f(X)$ in $\mathbb{R}_q$ will use this reduced form.

Note that n is a power of 2. In the context of CCA2-Secure CRYSTAL-Kyber, $n = 256$, which is 2^8. The choice of n is significant because it determines the structure of the polynomial rings $\mathbb{R}$ and $\mathbb{R}_q$.

$X^n + 1$ is the cyclotomic polynomial associated with the $2^{n'}$-th roots of unity. Cyclotomic polynomials are distinguished by having primitive roots of unity as their roots. Specifically, when $n' = 9$, $X^n + 1$ is associated with the 2^9-th roots of unity, indicating that its roots are the 512th roots of unity. Note that a cyclotomic polynomial itself doesn't satisfy $X^{512} = 1$. Instead, it has roots (let's call one of these roots z) that satisfy $z^{512} = 1$.

When we talk about the polynomial $X^n + 1$ being the $2^{(n')}$-th cyclotomic polynomial, we're referring to the polynomial whose roots are the primitive $2^{(n'+1)}$-th roots of unity with the following properties:

- $z^{512} = 1$: When the complex number z is raised to the power of 512, the result is 1.
- $z^k \neq 1$ for any positive integer k less than 512: The complex number z, when raised to any power less than 512, does not equal 1. In other words, it takes exactly 512 "multiplications" of z by itself to reach the value of 1, and no fewer.

So, the focus is on the roots of the polynomial and their properties, not the polynomial itself.

Vectors are assumed to be column vectors by default. The transpose of a vector $\mathbf{v}$ or a matrix $\mathbf{A}$ is denoted by $\mathbf{v}^T$ or $\mathbf{A}^T$, respectively. Note that a vector and a matrix are denoted as bold.

4.2.3 Modular Reduction

When computing modular reductions for a positive integer α, there are two types of reductions: **balanced** and **standard**.

For an even positive integer α, $r' = r \mod {}^{\pm}\alpha$ is defined as the unique element r' within the interval $-\frac{\alpha}{2} < r' \leq \frac{\alpha}{2}$. In the case of odd α, the range becomes $-\frac{\alpha-1}{2} \leq r' \leq \frac{\alpha-1}{2}$, ensuring that $r' = r \mod \alpha$.

For any positive integer α, we introduce $r' = r \mod {}^{+}\alpha$ as the distinct element r' within the range $0 \leq r' < \alpha$ so that $r' = r \mod \alpha$.

4.2.4 Rounding

For a given element $x \in \mathbb{Q}$, we use the notation $\lceil x \rfloor$ to denote the rounding of x to its nearest integer. Further, the $\lceil \cdot \rceil$ notation signifies rounding up to the nearest integer.[2] Finally, $\lfloor \cdot \rfloor$ rounds *down* to the nearest integer. Taking the number 4.3 as an instance:

$$\lceil 4.3 \rceil = 5$$
$$\lfloor 4.3 \rceil = 4$$

4.2.5 Compression and Decompression Functions

Definition 4.2.1 (Compression Function)**.** In CCA-Secure CRYSTAL-Kyber [13], authors define a compression function

$$\text{Compress}_q(x, d) = \lceil (2^d/q) \cdot x \rfloor \text{ mod}^+ \ 2^d \tag{4.2}$$

[2] Rounding up entails selecting the higher integer when a number is equidistant between two integers. This scenario arises when a number is equidistant from the two nearest integers.

which takes an element $x \in \mathbb{Z}_q$ as an input and returns an integer within the range $\{0, \ldots, 2^d - 1\}$, where $d < \lceil \log_2(q) \rceil$.

Hence, when using the compression function $\text{Compress}_q(x, d = 1)$, it takes an element $x \in \mathbb{Z}_q$ as input and maps it to an integer within the range $\{0, \ldots, 2^{d=1} - 1\}$, which is simply the set $\{0, 1\}$. Therefore, the result of $\text{Compress}_q(x, 1)$ will be either 0 or 1.

4.2.5.1 A Toy Example for the Compression Function

In the following, we present a concise demonstration of the compression function. Through this example, we aim to elucidate how the value x undergoes compression to yield $c = (53248)_{10}$.

Starting with the value

$$1234567890_{10} = 1001001100101100000001011010010_2,$$

which spans 31 digits, the compression process condenses it into a 16-bit representation. This compact representation retains the essence of the original value x, but with significantly reduced size.

```
1  def CompressionFunction(x, d):
2    c = (2**d // q) * x % (2**d)
3    return c
4
5  x = 1234567890
6  d = 16
7  c = CompressionFunction(x, d)
8
9  print(c)
```

Authors [14] also define a decompressing function such that

$$x' = \text{Decompress}_q(\text{Compress}_q(x, d), d),$$

where x' is an element in close proximity to x, more specifically,

$$|x' - x \bmod^{\pm} q| \leq B := \lceil \frac{q}{2^{d+1}} \rfloor.$$

The functions that fulfill these criteria are defined as:

$$\text{Compress}_q(x, d) = \lceil (2^d/q) \cdot x \rfloor \text{ mod}^+ \ 2^d$$

and

$$\text{Decompress}_q(x, d) = \lceil (q/2^d) \cdot x \rfloor.$$

4.2.5.2 Motivations Behind Compression and Decompression Functions

The main goal of using these functions is to **reduce the ciphertext size by discarding some low-order bits that do not significantly impact the probability of correct decryption**. In other words, the compression function is used to reduce the size of the ciphertext by discarding the least significant d bits of the value x. This results in a more efficient and compact representation of ciphertexts.

In the line 20 of Algorithm 5, the Decompress_q function is utilized to create error tolerance gaps by mapping the message bit 0 to 0 and 1 to $\lceil q/2 \rceil$ [1]. This process ensures that **there is a significant gap between the two possible values**, allowing for error correction during decryption.

Let's examine that. The particular line we are talking about is 20:

$$v := \text{NTT}^{-1}(\hat{\mathbf{t}}^T \circ \hat{\mathbf{r}}) + e_2 + \text{Decompress}_q(\text{Decode}_1(m), 1)$$

Here, the term $\text{Decompress}_q(\text{Decode}_1(m), 1)$ is responsible for creating the error tolerance gaps. Let's examine this in more detail. Recall that the Decompress_q function is defined as:

$$\text{Decompress}_q(x, d) = \lceil (q/2^d) \cdot x \rfloor.$$

When d is set to 1, as in the case of line 20, the function becomes

$$\text{Decompress}_q(x, 1) = \lceil (q/2) \cdot x \rfloor$$

Now, let's consider the two possible values of a single message bit, $m \in \{0, 1\}^1$.

When $m = 0$:

$$\text{Decompress}_q(0, 1) = \lceil (q/2) \cdot 0 \rfloor = 0$$

When $m = 1$:

$$\text{Decompress}_q(1, 1) = \lceil (q/2) \cdot 1 \rfloor = \lceil q/2 \rfloor$$

As you can see, the Decompress_q function maps the message bit 0 to 0 and the message bit 1 to $\lceil q/2 \rfloor$. This creates a gap between the two possible values, which helps ensure that errors can be tolerated and corrected during the decryption process.

4.2.6 Encoding and Decoding Functions

Decode$_l$ function converts a byte array B of size $32l$ into a polynomial $f = f_0 + f_1X + \cdots + f_{255}X^{255}$. Each f_i in the polynomial f is represented in l bits from the byte array B.

Mathematically, for each i from 0 to 255, we have:

$$f_i := \sum_{j=0}^{l-1} \beta_{il+j} * 2^j$$

where β_{il+j} represents the bits in the byte array B.

Algorithm 1 $\text{Decode}_l : B^{32l} \rightarrow \mathcal{R}_q$

1: **Input:** Byte array $B \in B^{32l}$
2: **Output:** Polynomial $f \in \mathcal{R}_q$
3: $(\beta_0, ..., \beta_{256l-1}) := \text{BytesToBits}(B)$
4: **for** i from 0 to 255 **do**
5: $\quad f_i := \sum_{j=0}^{l-1} \beta_{il+j} 2^j$
6: **end for**
7: **return** $f_0 + f_1X + f_2X^2 + \cdots + f_{255}X^{255}$

The **Encode**$_l$ function converts a polynomial f into a byte array of size $32l$. Each f_i in the polynomial f is converted to l bits and stored in the byte array.

Mathematically, the encoding process can be seen as the inverse operation of the decoding. For each f_i in f, we calculate l bits and store them in the byte array. Note that the factor l is indicative of the level of precision used in the encoding process. A larger l results in a larger byte array and subsequently higher precision.

4.2.7 Symmetric Primitives

Kyber's architecture incorporates a pseudorandom function, represented as PRF : $B_{32} \times B \rightarrow B^*$, along with an extendable output function denoted as XOF : $B^* \times B \times B \rightarrow B^*$. Additionally, Kyber utilizes two distinct hash functions, $H : B^* \rightarrow B_{32}$ and $G : B^* \rightarrow B_{32} \times B_{32}$, as well as a key-derivation function expressed as KDF : $B^* \rightarrow B^*$ [1].

4.2.8 Number-Theoretic Transform (NTT)

In this section, we will give a detailed information about the NTT transform used in CCA-Secure CYRSTALS-Kyber algorithm.

4.2.8.1 The 256th Roots of Unity for the Defining Polynomial $X^{256} + 1$

First, we should understand that $X^{256} + 1$ is the defining polynomial we're working with.

When this polynomial is set to 0, its solutions form the 256th roots of unity. The "**roots of unity**" are a concept from complex number theory that satisfy the equation $z^n = 1$. In other words, the roots of unity are complex numbers that, when raised to the power of 256, equal 1.

$$X^{256} + 1 = 0$$
$$X^{256} = -1$$

So, in our polynomial equation $X^{256} + 1 = 0$, the solutions are 256th roots of -1. So, when these complex numbers are raised to the power of 256, they will indeed equal -1.

For the polynomial $X^{256} + 1 = 0$, its roots are the 256th roots of -1, which are just as numerous and evenly distributed on the unit circle, but they are rotated half a unit counterclockwise compared to the 256th roots of 1.

4.2.8.2 How to Prove that the Defining Polynomial Has 256th Roots of Unity?

In the base field $\mathbb{Z}_{3329}$, we are looking for the existence of primitive n-th roots of unity.

It is a mathematical fact that a finite field (like $\mathbb{Z}_{q=3329}$) can have a primitive n-th root of unity if and only if n divides $q-1$. So, the existence of a primitive n-th root of unity in $\mathbb{Z}_q$ is tied to the factorization of $q-1$.

Given

$$\begin{aligned} q-1 &= 3329-1 \\ &= 2^8 \cdot 13, \end{aligned}$$

we can see that any number n that divides $2^8 * 13$ can have primitive n-th roots of unity in $\mathbb{Z}_q$. As 256 $(n = 2^8)$ divides $2^8 * 13$, there are primitive **256th roots of unity in** $\mathbb{Z}_q$. But 512 doesn't divide $2^8 * 13$, so there are no primitive 512th roots of unity in $\mathbb{Z}_q$.

Why does it matter? The existence of a primitive 256th root of unity but not a 512th means that when we try to find solutions to the equation $X^{256} + 1 = 0$ in $\mathbb{Z}_q$, we find 256 solutions.

4.2.8.3 Defining Polynomial $X^{256} + 1$ Factors into 128 Quadratic Polynomials

In the complex plane, an n-th root of unity refers to a complex number that, when raised to the power n, equals 1. That is, if $z^n = 1$, then z is an n-th root of unity. These roots lie on the unit circle in the complex plane, and they are evenly distributed around the circle. For the 256-th roots of unity, there will be 256 points on this unit circle.

Now, the defining polynomial we have is $X^{256} + 1 = 0$. The roots of this polynomial are the complex numbers that, when raised to the power 256, yield -1, not 1. These are different from the usual 256-th roots of unity; we can say they are "negated" 256th roots of unity.

Now, suppose we have a number, we'll call it ζ, that when raised to the power 256

gives -1 (hence a 256-th root of unity). This is a special type of number because it possesses a unique property: no matter how many times you square it, it will remain a 256-th root of unity. That's because $(-z)^2 = z^2$, so $(-1)^2 = 1$. That is, if ζ is a 256-th root of unity, so is ζ^2.

Using these special numbers, we can break our original polynomial into smaller chunks. These chunks look like $(X^2 - \zeta^{2i+1})$. Here i is just a counter that ranges from 0 to 127. This is where our 128 different pieces come from. Putting it all together, we can write:

$$X^{256} + 1 = (X^2 - \zeta) * (X^2 - \zeta^3) * (X^2 - \zeta^5) * \cdots * (X^2 - \zeta^{255}).$$

Each of these quadratic factors is of the form $(X^2 - \zeta^{2i+1})$, where ζ is a primitive 256-th root of unity modulo q.

$$X^{256} + 1 = \prod_{i=0}^{127}(X^2 - \zeta^{2i+1})$$

4.2.8.4 Performing NTT on Polynomial $f \in \mathbb{R}_q$: Representation as 128 First-Degree Polynomials

The Number Theoretic Transform (NTT) is a tool that's used for efficient polynomial multiplication. In this particular context, the NTT is being used to decompose the polynomial f with respect to the 128 distinct quadratic factors of the polynomial $X^{256} + 1$ (modulo q). Each of these quadratic factors is of the form $X^2 - \zeta^{2i+1}$, where ζ is a primitive 256th root of unity modulo q, and i ranges from 0 to 127.

In other words, we're taking our polynomial f, and seeing how it 'behaves' when we consider it relative to each of these 128 distinct factors.

So when we take $f \mod (X^2 - \zeta^{2i+1})$, we are dividing f by $X^2 - \zeta^{2i+1}$ and looking at the remainder. That's the "remainder of the polynomial f when divided by each quadratic factor" part. This remainder will be a polynomial of degree less than 2, i.e., a linear polynomial or a constant.

Hence, the NTT of f will be a vector of these 128 remainders. Each element of this vector is a polynomial of degree one (or a constant), and it represents the 'component'

of f relative to one of the quadratic factors of $X^{256} + 1$.

Thus, the NTT of $f \in \mathbb{R}_q$ can be written as the following.

$$(f \mod X^2 - \zeta^{2\cdot(0)+1}, \ldots, f \mod X^2 - \zeta^{2\cdot(127)+1}).$$

After you've taken your polynomial, $f \in \mathbb{R}_q$, and applied the Number Theoretic Transform (NTT), you get a vector of polynomials[3]. These polynomials are the remainders when f is divided by each of the 128 quadratic factors of the defining polynomial $X^{256} + 1$. This gives you the list (or vector): $(f \mod X^2 - \zeta^{2\cdot 0+1}, ..., f \mod X^2 - \zeta^{2\cdot 127+1})$. This is essentially seeing how f behaves when plugged into each of these quadratic equations.

Thus, the Number Theoretic Transform (NTT) transforms the polynomial f from the coefficient domain to the point-value domain, where each element of the result vector represents the value of the polynomial at a specific root of the defining polynomial.

To illustrate, let's consider an extremely simplified example with smaller numbers.

Let's say we have a defining polynomial $X^4 + 1$ which can be factored into $X^2 - i$ and $X^2 + i$ over the complex numbers, where i is the imaginary unit.

If we consider a polynomial $f = X^3 + 2X^2 + 3X + 4$, then the NTT of f would consist of the remainders of f when "divided" by each of these factors. So, we would compute $f \mod (X^2 - i)$ and $f \mod (X^2 + i)$, and the result would be a vector of two polynomials of degree one, which are the remainders of these operations.

In the context of $X^{256}+1$, we're doing the same thing, but with 128 different quadratic factors, and we're working modulo q, not over the complex numbers.

Each element of the output vector is the remainder of f when "divided" by each of these quadratic factors. This is essentially what's happening when the authors [1] say "**the NTT of a polynomial $f \in \mathbb{R}_q$ is a vector of 128 polynomials of degree one.**" It's a way to decompose f into simpler parts relative to the roots of the defining polynomial.

[3] In this context, when we say a "vector of polynomials," we don't mean it in the strict linear algebra sense of "vector," which might imply that we could perform certain operations like vector addition or scalar multiplication on these polynomials. Rather, we're using "vector" more loosely to simply mean a list or ordered collection of polynomials.

Next, this vector of linear (degree one) polynomials is converted into a form that is easier to handle computationally. This is what the phrase "*serialized to a vector in* $\mathbb{Z}_q^{256}$ *in the canonical way*" is referring to. This means that we view each linear polynomial $ax + b$ as two elements a and b in $\mathbb{Z}_q$ (the integers modulo q), and so the 128 polynomials are "flattened" into a single vector of 256 elements.

For this, the $NTT : \mathbb{R}_q \to \mathbb{R}_q$ is defined to be a bijection (a one-to-one correspondence) that maps f in $\mathbb{R}_q$ to a new polynomial that has the coefficient vector we mentioned above. This is essentially a representation change. You're changing the representation of the polynomial from being in the domain of $\mathbb{R}_q$ to a form that's easier to compute with.

The $NTT(f)$ is denoted as $\hat{f}$, and is represented as a polynomial of degree 255

$$\text{NTT}(f) = \hat{f} = \hat{f}_0 + \hat{f}_1 X + \ldots + \hat{f}_{255} X^{255},$$

with the coefficients defined as:

$$\hat{f}_{2i} = \sum_{j=0}^{127} X^{2j} f \cdot \zeta^{(2(i)+1)j}, \quad (4)$$

$$\hat{f}_{2i+1} \sum_{j=0}^{127} X^{2j+1} f \cdot \zeta^{(2(i)+1)j}. \quad (5)$$

There is one thing to note here. After the NTT is applied to f, the result is another polynomial we're calling $\hat{f}$. However, $\hat{f}$ is not just a simple polynomial; it's a collection of 128 polynomials, each of degree 1.

$$\text{NTT}(f) = \hat{f} = (\hat{f}_0 + \hat{f}_1 X, \hat{f}_2 + \hat{f}_3 X, \ldots, \hat{f}_{254} + \hat{f}_{255} X)$$

means that $\hat{f}$ is made up of 128 separate polynomials, each one looking like $\hat{f}_{2i} + \hat{f}_{2i+1} X$. In other words, the NTT of f is a complex object with 128 separate parts, each part being a degree-1 polynomial.

It's also important to note that this representation is purely algebraic, which means it's a mathematical construct used for the purposes of calculation and analysis. The actual polynomial $\hat{f}$ does not have any algebraic meaning on its own; its meaning comes from its role in the number theoretic transform and its relationship to the original polynomial f.

4.2.8.5 Efficient Multiplication of Two Polynomails with NTT.

$$f \cdot g = \text{INTT}(\text{NTT}(f) \circ \text{NTT}(g))$$

where the graph includes

- CT butterfly-based NTT point-wise multiplication [37], and
- GS butterfly-based INTT [37].

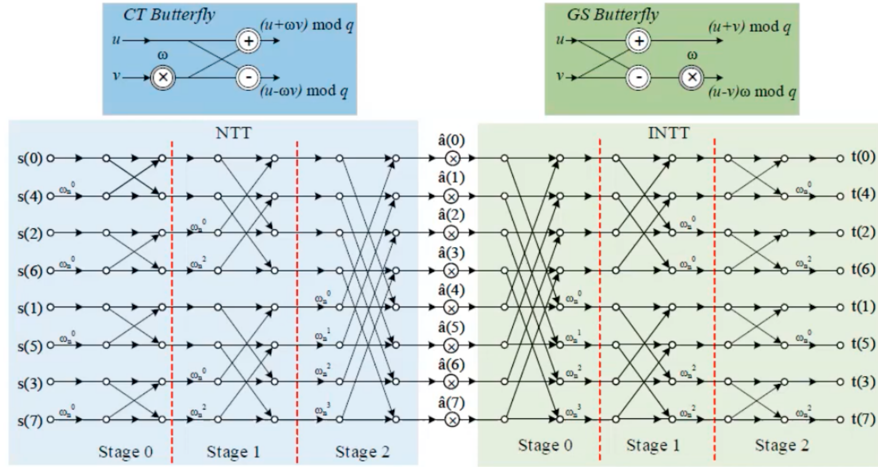


Figure 4.1: Polynomial Multiplication [37]

4.2.9 Uniform Sampling From the Ring of Polynomials $\mathbb{R}_q$

Uniform sampling in Kyber is a key process used to generate elements in $\mathbb{R}_q$. This is achieved via a deterministic approach employing the function **Parse** : $B^* \rightarrow \mathbb{R}_q$, which operates on an arbitrary input byte stream such that $B = b_0, b_1, b_2, \cdots \in B^*$.

This function computes the NTT-representation

$$\hat{a} = \hat{a}_0 + \hat{a}_1 x + \hat{a}_2 x^2 + ... + \hat{a}_{255} x^{255}$$

in $\mathbb{R}_q$ of $a \in \mathbb{R}_q$.

In other words, the Parse function is used to convert a byte stream into a polynomial in $\mathbb{R}_q$. Each byte (or a combination of bytes in this case) in the stream is used to form a coefficient of the polynomial. This process helps in transforming arbitrary

Algorithm 2 Parse: $B^* \to \mathbb{R}_{q=3329}^{n=256}$

1: **Input:** Byte stream $B = b_0, b_1, b_2 \ldots \in B^*$
2: **Output:** NTT-representation $\hat{a} \in \mathbb{R}_q$ of $a \in \mathbb{R}_q$
3: $i := 0$
4: $j := 0$
5: **while** $j < n$ **do**
6: $\quad d_1 := b_i + 256 \cdot (b_{i+1} \mod 16)$
7: $\quad d_2 := \lfloor b_{i+1}/16 \rfloor + 16 \cdot b_{i+2}$
8: $\quad$ **if** $d_1 < q$ **then**
9: $\quad\quad \hat{a}_j := d_1$
10: $\quad\quad j := j + 1$
11: $\quad$ **end if**
12: $\quad$ **if** $d_2 < q$ and $j < n$ **then**
13: $\quad\quad \hat{a}_j := d_2$
14: $\quad\quad j := j + 1$
15: $\quad$ **end if**
16: $\quad i := i + 3$
17: **end while**
18: **return** $\hat{a}_0 + \hat{a}_1 X + \ldots + \hat{a}_{n-1} X^{n-1}$

byte streams into a format (a polynomial in $\mathbb{R}_q$) that can be utilized effectively in the computations of the cryptosystem, such as polynomial multiplication in the NTT domain.

It's important to note that the use of uniform sampling is central to ensuring the statistical indistinguishability of the output from a truly random selection, an attribute that bolsters the cryptographic strength of CRYSTAL-Kyber.

4.2.10 Sampling from a Binomial Distribution

For *Central Binomial Distribution* (CBD) function in Kyber (See Lines 10 and 14 in Alg. 5) is used to generate the secret vector s and error (noise) vector **e** in the form of polynomials whose coefficients follow a centered binomial distribution. This dis-

tribution is chosen for its symmetric and mean-zero properties, which help to ensure that the noise doesn't bias the cryptographic computations.

The function takes as input a byte array B of length 64η. It then converts these bytes to bits. For each index from 0 to 255, it forms two groups of η bits, interpreted as integers to give a and b. The difference between a and b gives the coefficient f_i of the polynomial. Thus, each f_i is a random variable following the centered binomial distribution.

Finally, CBD outputs a polynomial f which is a sum of f_i times corresponding powers of X. This polynomial is used as noise in the cryptosystem. This ensures that every coefficient in the polynomial f is derived deterministically from 64_η bytes of pseudorandom function output, following the binomial distribution 64_η.

Algorithm 3 CBD$_\eta$: $B^{64\eta} \rightarrow \mathbb{R}_q$

Input: Byte array $B = (b_0, b_1, ..., b_{64\eta-1}) \in B^{64\eta}$
Output: Polynomial $f \in R_q$
1: $(\beta_0, ..., \beta_{512\eta-1}) \leftarrow \text{BytesToBits}(B)$
2: **for** i from 0 to 255 **do**
3: $\quad a \leftarrow \sum_{j=0}^{\eta-1} \beta_{2i\eta+j}$
4: $\quad b \leftarrow \sum_{j=0}^{\eta-1} \beta_{2i\eta+\eta+j}$
5: $\quad f_i \leftarrow a - b$
6: **end for**
7: **return** $f_0 + f_1X + f_2X^2 + \ldots + f_{255}X^{255}$

4.3 Kyber.CPAPKE.KeyGen() Algorithm

In this section, a detailed examination of Kyber.CPAPKE.KeyGen() will be provided. The pseudo-code of the algorithm is directly taken from the Supporting Documentation of the CRYSTAL-Kyber [1].

Algorithm 4 Kyber.CPAPKE.KeyGen()

Input: None

Output: Secret key $sk \in \mathcal{B}^{12 \cdot k \cdot n/8}$

Output: public key $pk \in \mathcal{B}^{12 \cdot k \cdot n/8+32}$

1: $d \leftarrow \mathcal{B}^{32}$
2: $(\rho, \sigma) \leftarrow \mathrm{G}(d)$ ▷ Generate ρ and σ
3: $N \leftarrow 0$ ▷ Initialize N
4: **for** $i = 0$ to $k-1$ **do** ▷ Generate the matrix $\hat{\mathbf{A}} \in \mathbb{R}_q^{k \times k}$ in NTT domain
5:     **for** $j = 0$ to $k-1$ **do**
6:         $\hat{\mathbf{A}}[i][j] \leftarrow \mathrm{Parse}(\mathrm{XOF}(\rho, j, i))$
7:     **end for**
8: **end for**
9: **for** $i = 0$ to $k-1$ **do**
10:     $\mathbf{s}[i] \leftarrow \mathrm{CBD}\eta_1(\mathrm{PRF}(\sigma, N))$ ▷ Sample $\mathbf{s} \in \mathbb{R}_q^k$ from $B\eta_1$
11:     $N \leftarrow N + 1$
12: **end for**
13: **for** $i = 0$ to $k-1$ **do**
14:     $\mathbf{e}[i] \leftarrow \mathrm{CBD}\eta_1(\mathrm{PRF}(\sigma, N))$ ▷ Generate $\mathbf{e} \in \mathbb{R}_q^k$ from $B\eta_1$
15:     $N \leftarrow N + 1$
16: **end for**
17: $\hat{\mathbf{s}} \leftarrow \mathrm{NTT}(\mathbf{s})$ ▷ Perform NTT
18: $\hat{\mathbf{e}} \leftarrow \mathrm{NTT}(\mathbf{e})$ ▷ Perform NTT
19: $\hat{\mathbf{t}} \leftarrow \hat{\mathbf{A}} \circ \hat{\mathbf{s}} + \hat{\mathbf{e}}$ ▷ Calculate $\hat{t}$
20: $pk \leftarrow (\mathrm{Encode}_{12}(\hat{\mathbf{t}} \bmod q) \parallel \rho)$ ▷ Encode public key
21: $sk \leftarrow \mathrm{Encode}_{12}(\hat{\mathbf{s}} \bmod q)$ ▷ Encode secret key
22: **return** (pk, sk)

Let's explain the algorithm line by line.

- **Line 1 & 2:**

The function $G : B^* \rightarrow B^{32} \times B^{32}$ is a *cryptographic hash function* that takes an input from the set of all binary strings B^* and generates an output which is a pair of

32-byte binary strings. In other words, the output is a 2-tuple, where each element is a binary string of length 32 bytes.

In the context of the CRYSTALS-Kyber cryptographic protocol, G yields two distinct 32-byte sequences, denoted as ρ and σ. This function G is typically instantiated with SHA3-512 [1], a cryptographic hash function that provides a 64-byte (or 512-bit) result.

Mathematically, given a binary input d to the function G, it can be expressed as:

$$G(d) = \text{SHA3-512}(d)$$

This results in a 64-byte output. Subsequently, this output is divided into two 32-byte outputs:

$$(\rho, \sigma) = (g[0:32], g[32:64])$$

In this arrangement, if G is implemented as SHA3-512, ρ becomes the first 32 bytes of the SHA3-512 hash of d, and σ becomes the remaining 32 bytes.

Within the Kyber protocol, these outputs serve different functions. The seed ρ is utilized by the XOF function to create pseudorandom elements of the public matrix A (Refer to Line 6), while the seed σ is employed by the PRF function to generate the secret key $\mathbf{s}$ and the error vector $\mathbf{e}$ (Refer to Line 10).

- **Line 4, 5, 6, 7 & 8:**

In the given part of the algorithm, we're constructing $k \times k$ matrices $\hat{\mathbf{A}} \in \mathbb{R}_q^{k \times k}$ (exactly k^2 of them) with entries in the polynomial ring $\mathbb{R}_q$.

$\mathbb{R}_q^{k \times k}$ denotes a $k \times k$ matrix where each element is a polynomial from $\mathbb{R}_q$.

Notice how the k denotes how many matrices that the algorithm is going to use.

$$\begin{pmatrix} A_{0,0}(X) & & \cdots & A_{0,k-1}(X) \\ & & \cdots & \\ \vdots & \vdots & \ddots & \vdots \\ A_{k-1,0}(X) & & \cdots & A_{k-1,k-1}(X) \end{pmatrix} \cdot \begin{pmatrix} s_0(X) \\ s_1(X) \\ \vdots \\ s_{k-1}(X) \end{pmatrix} + \begin{pmatrix} e_0(X) \\ e_1(X) \\ \vdots \\ e_{k-1}(X) \end{pmatrix} = \begin{pmatrix} t_0(X) \\ t_1(X) \\ \vdots \\ t_{k-1}(X) \end{pmatrix}$$

Each of these matrices ($\hat{\mathbf{A}}$) are in the form of $\mathbb{R}_q^{k\times k}$ in NTT domain. The statement that the matrix is in the Number Theoretic Transform (NTT) domain suggests that the polynomials filling this matrix have undergone an NTT. This makes computations more efficient while preserving cyclic convolution properties, which are crucial in polynomial multiplication, an essential operation in the Kyber protocol.

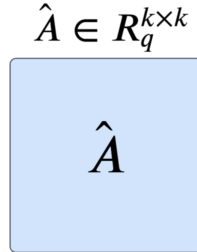


Figure 4.2: Generating the $\mathbf{A}$ in the KeyGen() Algorithm [13].

$$\begin{pmatrix}
a_{0,0} & -a_{k-1,0} & -a_{k-2,0} & \cdots & -a_{1,0} \\
a_{1,0} & a_{0,0} & -a_{k-1,0} & \cdots & \\
a_{2,0} & a_{1,0} & a_{0,0} & \cdots & \\
a_{3,0} & a_{2,0} & a_{1,0} & \cdots & \ddots \\
\vdots & \vdots & & \vdots & a_{0,0} \\
a_{k-2,0} & & & & \\
a_{k-1,0} & a_{k-2,0} & a_{k-3,0} & \cdots & a_{0,0}
\end{pmatrix}$$

The values of each entry in this matrix, $\hat{A}^T[i][j]$, are determined by an Extendable Output Function (XOF : $B^* \times B \times B \to B^*$) applied on the seed ρ, and the indices i and j.

XOF is a type of cryptographic hash function that can produce output of arbitrary length, unlike traditional hash functions that have a fixed output length. This makes XOFs particularly useful in applications that require variable-length output.

In the Support Document [1], it is written that XOF is instantiated with SHAKE-128 algorithm. SHAKE-128 is a specific instance of an XOF. It is part of the SHA-3 family of cryptographic hash functions, which were selected as the winners of the NIST hash function competition. The name "SHAKE" stands for "Secure Hash Algorithm and KEccak," reflecting its origins in the Keccak algorithm.

- **Line 9, 10, 11 & 12:**

This piece of the algorithm refers to the generation of the secret key s.

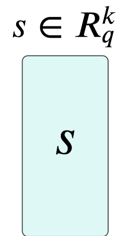


Figure 4.3: Generating the s in the KeyGen() [13].

- **Line 13, 14, 15 & 16:**

This piece of the algorithm refers to the generation of the error e.

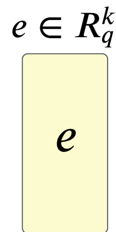


Figure 4.4: Generating the e in the KeyGen() [13].

- **Line 17, 18 & 19:**

In this piece of the code, Kyber.CPAPKE.KeyGen() performs an NTT transformation to the vectors $\mathrm{s} \in \mathbb{R}_q^k$ and $\mathrm{e} \in \mathbb{R}_q^k$.

Notice that the matrix $\hat{\mathrm{A}}$ and $\hat{\mathrm{s}}$ gets polynomially multipled in the NTT domain and the error vector $\hat{\mathrm{e}}$ is added to the result. As the reader can see, this is the core of the *Learning with Error* problem.

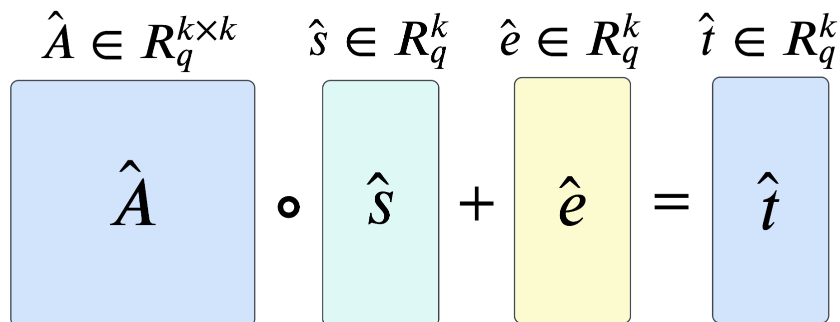


Figure 4.5: Generating the (pk, sk) in the KeyGen() [13].

- **Line 20, 21 & 22:**

In the end, we have a public key $pk := \mathbf{As} + \mathbf{e} = \mathbf{t}$, and a secret key $sk := \mathbf{s}$.

You can see this as one part's (let us say Alice) generating a secret key sk, and the public key pk. The public key is then sent to Bob. In Kyber, the secret key sk is represented as a byte array, and its length is determined by the parameters k and n.

Here, k represents the **number of polynomials in the secret key**, and n is the **degree of the polynomials**. The factor 12 comes from the encoding used in Kyber, where each element in the polynomial takes up 12 bits (1.5 bytes).

4.3.1 Length of the Secret and Public Keys

Before closing this section, it is pivotal to understand the structure and the rationale behind the lengths of the secret key, sk, and the public key, pk. Here is a succinct breakdown of the lengths of these keys and the reasoning behind them:

- **Secret Key sk Length:**
  - Comprised of a vector with k polynomials.
  - A polynomial in Kyber has 256 coefficients taken from $\mathbb{Z}_{3329}$.
  - Each coefficient is 12-bits long, due to being derived modulo 3329. The Boolean representation of any number taken from this coefficient domain $[0, q-1]$ is denoted by 12-bits.
  - Consequently, the length of sk is calculated as:

    $$\text{Length of } sk = \frac{12 \cdot n \cdot k}{8} \text{ bytes}$$

    where $n = 256$ represents the number of coefficients in each polynomial.
- **Public Key pk Length:**
  - Structurally similar to sk but is concatenated with an additional 32-byte entity ρ.
  - ρ is a 32-byte binary string derived from the function G.

– Thus, the length of pk is given by:

$$\text{Length of } pk = \left(\frac{12 \cdot k \cdot n}{8}\right) + 32 \text{ bytes}$$

which accounts for the encapsulated information essential for secure cryptographic communication.

4.4 Kyber.CPAPKE.Enc(pk, m, r)

This section presents a detailed analysis of the Kyber.CPAPKE.Enc() algorithm, a central element of the CRYSTALS-Kyber cryptographic protocol. The pseudo-code of the algorithm is directly taken from the Supporting Documentation of the Kyber [1].

Let us examine the Alg. 5 line by line.

- **Line 2:**

Remember the Line 19 and Line 20 of the Kyber.CPAPKE.KeyGen() Algorithm 4. This is where Alice generates her public and secret key-pair (pk, sk).

$$\hat{\mathbf{t}} \leftarrow \hat{\mathbf{A}} \circ \hat{\mathbf{s}} + \hat{\mathbf{e}} \tag{4.3}$$

$$pk \leftarrow (\text{Encode}_{12}(\hat{\mathbf{t}} \bmod q) \parallel \rho) \tag{4.4}$$

Thus, upon receiving Alice's public key 'pk', Bob decodes this into a polynomial vector $\hat{\mathbf{t}}$ in the Number Theoretic Transform (NTT) domain.

$$\hat{\mathbf{t}} \leftarrow \text{Decode}_{12}(pk) \tag{4.5}$$

- **Line 3:**

To obtain the seed ρ that Alice used in generating her public key, Bob uses the public key that Alice shared with him, denoted as

$$\rho = pk + 12 \cdot k \cdot (n/8). \tag{4.6}$$

- **Line 4, 5, 6, 7 and 8:**

This ρ serves as input to the Extendable Output Function (XOF), which is instantiated using SHAKE-128 according to the FIPS-202 standard in the context of Kyber (as discussed earlier in the explanation of the Algorithm 4).

Through this process, Bob is able to generate the same matrix $\hat{\mathbf{A}}$ as Alice initially did, using the same ρ value and the consistent instantiation of the XOF with SHAKE-128.

- **Line 9, 10, 11, and 12:**

To introduce randomness into the process (which enhances security), Bob generates a random vector $\mathbf{r} \in \mathbb{R}_q^k$ that follows a discrete binomial distribution B_{η_1}. The byte string '$r \in B^{32}$' is used as a seed (or let's say random coins) to generate this vector of polynomials. The inclusion of such randomness ensures that the encryption process yields different ciphertexts even for the same plaintext, strengthening the overall security.

- **Line 13, 14, 15 & 16:**

In the context of Kyber encryption, the error vector $\mathbf{e}_1 \in \mathbb{R}_q^k$ is generated by drawing samples from a centered binomial distribution $B\eta_2$, defined as follows:

- For a fixed η, we sample $(a_1, ..., a_\eta, b_1, ..., b_\eta)$ uniformly at random from the binary set $\{0, 1\}^{2\eta}$.
- We then compute the difference between the sum of the a_i and the sum of the b_i, i.e., $\sum_{i=1}^{\eta} a_i - \sum_{i=1}^{\eta} b_i$.
- This sampling operation is represented as CBD$_{\eta_2}$, and each of its outcome will be a number in the set $-\eta, -\eta + 1, \cdot, \eta - 1, \eta$, and is symmetric around 0.
- For each element of the error vector $\mathbf{e}_1$, we run the CBDη_2 operation using the pseudorandom function (PRF) applied to the random byte string $r \in B^{32}$ and a nonce N.

It's important to note that η is set to either 2 or 3, depending on the security level of the Kyber variant in use.

In a sense, each entry in the vector $\mathbf{e}_1$ is the result of a "random walk" process where each step is either forward (counted as positive) or backward (counted as negative), each with equal probability. The total number of steps is equal to η. The use of the binomial distribution for this purpose is due to its desirable cryptographic properties, such as its symmetry, and the fact that it approximates the Gaussian distribution for large η while being easier to sample from in a cryptographic setting.

- **Line 17:**

The difference between $\mathbf{e}_1$ and e_2 arises from their roles in the encryption scheme, which demands different types of variables.

$\mathbf{e}_1$ is a vector because it's used in the matrix-vector multiplication $\hat{\mathbf{A}^T} \circ \hat{\mathbf{r}}$ during the encryption process. Specifically, it's added to the result of this multiplication. The multiplication of a matrix with a vector results in a vector, hence $\mathbf{e}_1$ must be a vector of the same dimension to make the addition possible. Each element of $\mathbf{e}_1$ vector is generated by calling the CBDη_2 function with the PRF applied to r and N. This is repeated for $i = 0$ to $k-1$, hence generating k elements, which forms a k-dimensional vector $\mathbf{e}_1 \in R_q^k$.

The polynomial e_2 on the other hand, is used directly in the computation of $\mathbf{v}$ and hence it doesn't need to be a vector. The computation of $\mathbf{v}$ involves polynomial multiplications and additions where e_2 is added to the result. In this case, e_2 is a single polynomial sampled from the same CBD$_{\eta_2}$ function with the PRF applied to r and N.

In summary, the necessity for $\mathbf{e}_1$ to be a vector and e_2 to be a polynomial arises naturally from their respective roles in the encryption process.

- **Line 18:**

Line 18 applies the Number Theoretic Transform (NTT) to the randomization vector $\mathbf{r} \in \mathbb{R}_q^k$.

The reason $r \in \mathbb{R}_q^k$ is transformed via the NTT before proceeding with further computations is because the subsequent operations (specifically, the multiplications in line

19 and 20) require its operands to be in the NTT domain to exploit the speedup in polynomial multiplication.

The application of the NTT to $\mathbf{r}$ results in $\hat{\mathbf{r}}$. This transformed version of $\mathbf{r}$, i.e., $\hat{\mathbf{r}}$, is in the NTT domain, which allows for efficient polynomial multiplications with other polynomials in the NTT domain (like $\hat{\mathbf{A}}$ and $\hat{\mathbf{t}}$ in this case).

- **Line 19, 20, 21, 22 & 23:**

In the next phase of the encryption process, Bob calculates the vectors $\mathbf{u}$ and $\mathbf{v}$.

The process begins with Bob performing a fast polynomial multiplication, which is effectively multiplication in the Number Theoretic Transform (NTT) domain, between the matrix $\hat{\mathbf{A}}$ and the vector $\hat{\mathbf{r}}$. After this multiplication, Bob applies the Inverse Number Theoretic Transform (INTT) to the result to bring the data back into the coefficient domain.

Once in the coefficient domain, Bob adds a noise vector $\mathbf{e}_1 \in \mathbb{R}_q^k$ drawn from the distribution $B\eta_2$ to introduce some randomness. This forms the vector $\mathbf{u}$, adding a level of security to the encryption process.

Meanwhile, Bob computes $\mathbf{v}$ by multiplying $\hat{\mathbf{t}}^T$ with $\mathbf{r}$, adding another noise term (e_1 also from B_{η_1}), and adding the decompressed message m. Consequently, the vector $\mathbf{v}$ embeds the actual message m, masked within the noise and complexity of polynomial arithmetic.

$$v \leftarrow \text{NTT}^{-1}(\hat{\mathbf{t}}^T \circ \hat{\mathbf{r}}) + e_2 + \text{Decompress}_q(\text{Decode}_1(m), 1) \tag{4.7}$$

$$(v := \mathbf{t}^T \cdot \mathbf{r} + e_2 + \text{Decompress}_q(m, 1)) \tag{4.8}$$

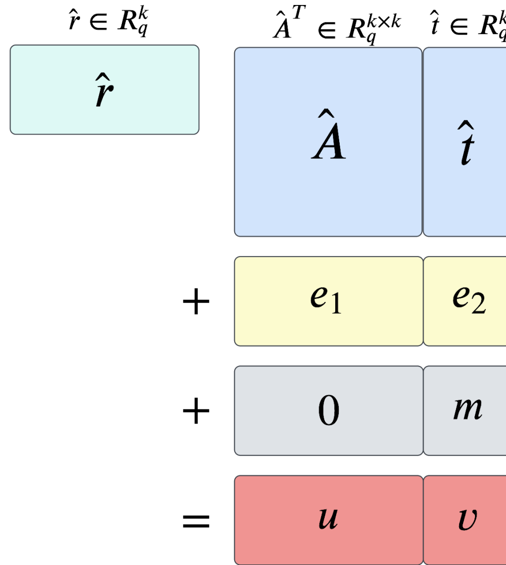


Figure 4.7: Bob Generating the $\mathbf{u}$ and v [13].

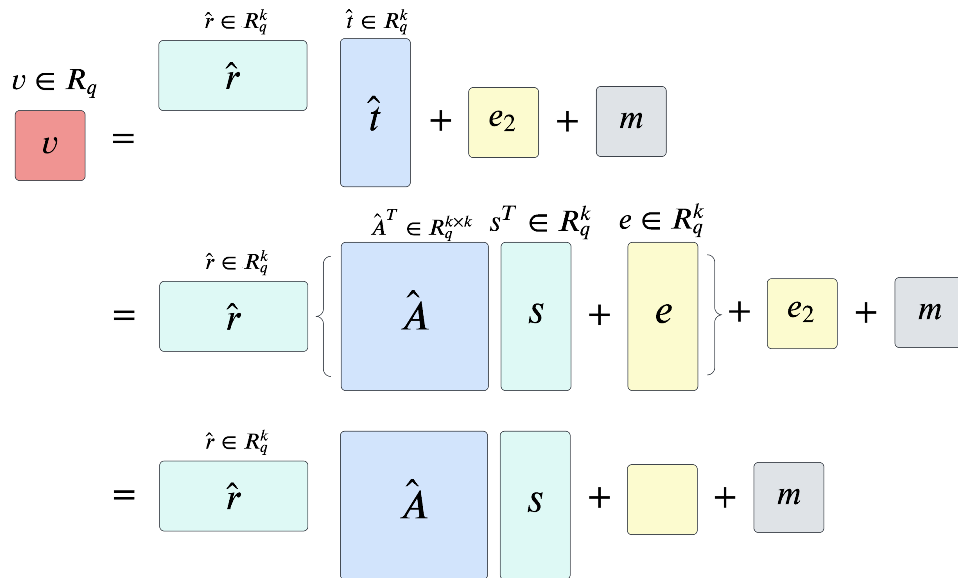


Figure 4.6: How Does Bob Computes the polynomial $v \in \mathbb{R}_q$ [13]?

The overall of the Encryption Algorithm can be fut into a figure as follows.

Both $\mathbf{u}$ and $\mathbf{v}$ are then compressed and encoded, producing two components of the ciphertext, c_1 and c_2. This compression reduces the size of these vectors of polynomials and, consequently, the overall ciphertext size. The compression operation is facilitated by the Compress_q function, which converts elements in $\mathbb{Z}_q$ into integers within the range $\{0, \ldots, 2^d - 1\}$.

Finally, the ciphertext c is formed by concatenating c_1 and c_2. Bob sends c to Alice, which represents the encrypted form of his message.

Algorithm 5 Kyber.CPAPKE.Enc(pk, m, r): encryption [14]

Input: Public key $pk \in \mathcal{B}^{12 \cdot k \cdot n/8+32}$

Input: Message $m \in \mathcal{B}^{32}$

Input: Random coins $r \in \mathcal{B}^{32}$

Output: Ciphertext $c \in \mathcal{B}^{d_u \cdot k \cdot n/8 + d_v \cdot n/8}$

1: $N \leftarrow 0$

2: $\hat{\mathbf{t}} \leftarrow \text{Decode}_{12}(pk)$

3: $\rho \leftarrow pk + 12 \cdot k \cdot (n/8)$

4: **for** i from 0 to $k-1$ **do** ▷ Generate the matrix $A^T \in \mathbb{R}_q^{k \times k}$ in NTT domain

5: **for** $j = 0$ to $k-1$ **do**

6: $\hat{\mathbf{A}}^T[i][j] \leftarrow \text{Parse}(\text{XOF}(\rho, i, j))$

7: **end for**

8: **end for**

9: **for** $i = 0$ to $k-1$ **do**

10: $\mathbf{r}[i] \leftarrow \text{CBD}_{\eta_1}(\text{PRF}(r, N))$ ▷ Sample $\mathbf{r} \in \mathbb{R}_q^k$ from B_{η_1}

11: $N \leftarrow N + 1$

12: **end for**

13: **for** $i = 0$ to $k-1$ **do**

14: $\mathbf{e}_1[i] \leftarrow \text{CBD}_{\eta_2}(\text{PRF}(r, N))$ ▷ Generate the error vector $\mathbf{e}_1 \in \mathbb{R}_q^k$ from B_{η_2}

15: $N \leftarrow N + 1$

16: **end for**

17: $e_2 \leftarrow \text{CBD}_{\eta_2}(\text{PRF}(r, N))$ ▷ Generate the error polynomial e_2 $\mathbb{R}_q$

18: $\hat{\mathbf{r}} \leftarrow \text{NTT}(r)$ ▷ Perform NTT

19: $\mathbf{u} \leftarrow \text{NTT}^{-1}(\hat{\mathbf{A}}^T \circ \hat{\mathbf{r}}) + \mathbf{e}_1$ ▷ $\mathbf{u} := \mathbf{A}^T \cdot \mathbf{r} + \mathbf{e}_1$

20: $v \leftarrow \text{NTT}^{-1}(\hat{\mathbf{t}}^T \circ \hat{\mathbf{r}}) + e_2 + \text{Decompress}_q(\text{Decode}_1(m), 1)$ ▷ $v := \mathbf{t}^T \cdot \mathbf{r} + e_2 + \text{Decompress}_q(m, 1)$

21: $c_1 \leftarrow \text{Encode}_{d_u}(\text{Compress}_q(\mathbf{u}, d_u))$ ▷ Compress $\mathbf{u}$

22: $c_2 \leftarrow \text{Encode}_{d_v}(\text{Compress}_q(v, d_v))$ ▷ Compress v

23: $c \leftarrow (c_1 \parallel c_2)$ ▷ Concatenate c_1 and c_2

24: **return** c ▷ $c \leftarrow (\text{Compress}_q(\mathbf{u}, d_u), \text{Compress}_q(v, d_v))$

4.5 Kyber.CPAPKE.Dec(c, sk)

Algorithm 6 Kyber.CPAPKE.Dec(c, sk): decryption [14]

Input: Secret key $sk \in B^{12 \cdot k \cdot n/8}$

Input: Ciphertext $c \in B^{d_u \cdot k \cdot n/8 + d_v \cdot n/8}$

Output: Message $m \in B^{32}$

1: $\mathbf{u} \leftarrow \text{Decompress}_q(\text{Decode}_{d_u}(c), d_u)$

2: $v \leftarrow \text{Decompress}_q(\text{Decode}_{d_v}(c + d_u \cdot k \cdot n/8), d_v)$

3: $\hat{\mathbf{s}} \leftarrow \text{Decode}_{12}(sk)$

4: $m \leftarrow \text{Encode}_1(\text{Compress}_q(v - \text{NTT}^{-1}(\hat{\mathbf{s}}^T \circ \text{NTT}(\mathbf{u})), 1))$ ▷ $m \leftarrow \text{Compress}_q(v - \mathbf{s}^T\mathbf{u}, 1)$

5: **return** m

Let us explain the algorithm line by line.

- Line 1 and 2:

Remember Lines 21 and 22 of Algorithm 5, where Bob compresses the vector of polynomials $\mathbf{u}$ and the polynomial v into c_1 and c_2, respectively. These are the two parts that are later concatenated into the ciphertext c.

$$c_1 \leftarrow \text{Encode}_{d_u}(\text{Compress}_q(\mathbf{u}, d_u)) \qquad \text{(Compress } \mathbf{u}\text{)}$$
$$c_2 \leftarrow \text{Encode}_{d_v}(\text{Compress}_q(v, d_v)) \qquad \text{(Compress } v\text{)}$$

Thus, as these are compressed, Alice decompresses them into, again, $\mathbf{u}$ and v.

- Line 3:

In this step, Alice decodes the byte-array secret key as into, again, NTT domain polynomial. This is necessary as in the Line 4, we need to perform an efficient polynomial multiplication.

- Line 4:

This is where the decryption happens.

Figure 4.8: Decryption of Message m [13].

Let us see how the algorithm actually works.

Figure 4.9: Explanation of the Decryption Phase Done by Alice [13].

4.6 Kyber.CCAKEM.KeyGen()

Kyber.CCAKEM (IND-CCA2 [1]) is constructed from the IND-CPA-secure public-key encryption scheme through Fujisaki-Okamoto [21] transform.

Algorithm 7 Kyber.CCAKEM.KeyGen()

Input: None

Output: Public key $pk \in \mathcal{B}^{12 \cdot k \cdot n/8+32}$, secret key $sk \in \mathcal{B}^{24 \cdot k \cdot n/8+96}$

1: $z \leftarrow \mathcal{B}^{32}$

2: $(pk, sk') \leftarrow$ Kyber.CPAPKE.KeyGen() ▷ Generate public and secret keys

3: $sk \leftarrow (sk' \parallel pk \parallel H(pk) \parallel z)$ ▷ Concatenate and hash

4: **return** (pk, sk)

4.6.1 Length of the Secret and Public Keys

Given the Kyber.CCAKEM.KeyGen() algorithm, we can break down the lengths of sk and pk as follows, considering the constructions from the CPAPKE.KeyGen() algorithm and the subsequent concatenations:

- The secret key sk' is a vector of k polynomials, each containing n coefficients. Since each coefficient is represented modulo 3329, it is 12-bits long. Therefore, the length of sk in bytes is:

$$\text{Length of } sk' = \frac{12 \cdot k \cdot n}{8}$$

- The public key pk remains the same as in Kyber.CPAPKE.KeyGen(), comprising a vector of k polynomials each having n coefficients, along with a 32-byte component, ρ. Thus, its length is:

$$\text{Length of } pk = \left(\frac{12 \cdot k \cdot n}{8}\right) + 32$$

- The secret key sk in Kyber.CCAKEM.KeyGen(), denoted as sk' in the algorithm, is a concatenation of the original secret key from the CPA-secure KeyGen() algorithm, the public key pk, the hash of the public key $H(pk)$, and a 32-byte string z. Given that $H(pk)$ is instantiated with SHA3-256, it has a fixed length of 32 bytes. Therefore, the length of sk is:

$$\text{Length of } sk = \left(\frac{12 \cdot k \cdot n}{8}\right) + \left(\frac{12 \cdot k \cdot n}{8}\right) + 32 + 32 + 32$$

$$\text{Length of } sk = \left(\frac{24 \cdot k \cdot n}{8}\right) + 96$$

In conclusion, these lengths ensure the incorporation of all necessary components in sk and pk, providing sufficient information and security features required by the Kyber.CCAKEM cryptographic scheme.

4.7 Kyber.CCAKEM.Encaps(pk, m, r)

Secondly, we have the *encapsulation* algorithm [1], which encapsulates the secret message $m \in \mathcal{B}^{32}$ using the public key pk[4] such that only the party who has possession of the corresponding secret key to decapsulate the message $c \leftarrow (c_1 \parallel c_2)$.

The message m is later used by both parties to drive an *ephemeral session key* $\mathbf{K}$. In general, emphemeral keys are **not masked** as decapsulation is performed using the long-term secret key.

Algorithm 8 Kyber.CCAKEM.Enc(pk)

Input: Public key $pk \in \mathcal{B}^{12 \cdot k \cdot n/8+32}$

Output: Ciphertext $c \in \mathcal{B}^{d_u \cdot k \cdot n/8 + d_v \cdot n/8}$, shared key $K \in \mathcal{B}^*$

1: $m \leftarrow \mathcal{B}^{32}$ ▷ Generate a random message m
2: $m \leftarrow \mathrm{H}(m)$ ▷ Hash m
3: $(\overline{K}, r) \leftarrow \mathrm{G}(m \parallel \mathrm{H}(pk))$ ▷ Generate $\overline{K}$ and r
4: $c \leftarrow$ Kyber.CPAPKE.Enc(pk, m, r) ▷ Encrypt message m
5: $K \leftarrow \mathrm{KDF}(\overline{K} \parallel \mathrm{H}(c))$ ▷ Derive shared key K
6: **return** (c, K)

Note that in the Support Document, the hash function H is instantiated with SHA3-256 [1].

4.8 Kyber.CCAKEM.Enc(pk)

The algorithm Kyber.CCAKEM.Enc(pk) [1] is meticulously designed to achieve secure key encapsulation, which is pivotal for resisting against various cryptographic attacks, specifically, chosen-ciphertext attacks.

[4] https://www.youtube.com/watch?v=R6lNR3ihVuU&t=363s

Algorithm 9 Kyber.CCAKEM.Enc(pk): encryption [1]

Input: Public key $pk \in \mathcal{B}^{12 \cdot k \cdot n/8+32}$

Output: Ciphertext $c \in \mathcal{B}^{du \cdot k \cdot n/8+d_v \cdot n/8}$, Shared key $K \in \mathcal{B}^*$

1: $m \leftarrow \mathcal{B}^{32}$
2: $m \leftarrow H(m)$
3: $(\bar{K}, r) := G(m \parallel H(pk))$
4: $c :=$ Kyber.CPAPKE.Enc(pk, m, r)
5: $K :=$ KDF$(\bar{K} \parallel H(c))$
6: **return** (c, K)

- The hashing of m using the SHA3-256 hash function ensures the generation of a fixed-size, seemingly random string from the plaintext m, augmenting the security by mitigating the risk of exposure of the original plaintext.

- The concatenation of m and $H(pk)$ and the subsequent application of the function G facilitate the creation of $\bar{K}$ and r, incorporating both the hashed plaintext and the hashed public key, adding an additional layer of security and integrity to the derived shared secret.

- The ciphertext c is encrypted using Kyber.CPAPKE.Enc, incorporating the public key, plaintext, and the random value r, ensuring the encapsulation of the secret.

- The Key Derivation Function (KDF) takes as input the concatenated $\bar{K}$ and the hash of the ciphertext c, processed using SHAKE-256, resulting in the final session key K. This step is crucial as it ensures that the final session key K is influenced by both the secret plaintext and the observed ciphertext, fortifying the scheme against potential alterations by an attacker.

In this algorithm, $\bar{K}$ is integral and serves as a blinded version of the shared secret. The term "blinded" here implies that while $\bar{K}$ is derived partly from a secret (the plaintext m), it does not serve directly as the session key. This blinding is essential for maintaining the secrecy of the session key K, even when some information about $\bar{K}$ is exposed.

Moreover, the verification of the integrity of the ciphertext is inherently built into the decryption process. If an adversary tampers with the ciphertext c, the decryption process will derive a different $\bar{K}'$ and subsequently a different c', invoking the fallback mechanism in the decapsulation process. This mechanism alerts the receiver to the discrepancy, reinforcing the resilience of the algorithm against illicit modifications and chosen-ciphertext attacks. In essence, the intricate design of this algorithm, the utilization of secure hash functions such as SHA3-256 and SHAKE-256, and the meticulous construction of the keys $\bar{K}$ and K underscore the robustness and security provided by the Kyber.CCAKEM.Enc(pk) in the realm of post-quantum cryptography.

4.9 Kyber.CCAKEM.Decaps(c, sk)

The Kyber.CCAKEM.Decaps(c, sk) algorithm is used to decapsulate the ciphertext c using the long term secret key [1].

Kyber.CCAKEM.Decaps(c, sk) algorithm happened to be the Fujisaki-Okamoto transform of the Kyber.CPAKEM.Dec(c, sk) algorithm to provide IND-CCA2 security. The FO transformation ([21], [26]) needed for CRYSTALS-Kyber involves not only utilizing Algorithms 4, 5, and 6 but also employing two distinct hash functions, H (refer to Line 8) and G (refer to Line 5), along with a key derivation function (KDF). Ones who want to learn more about these hash functions are encouraged to visit the Kyber's support document [1].

The core concept of this transformation is to verify the legitimacy of the output (ciphertext) of the Kyber.CPAPKE.Dec():

$$m' \leftarrow \text{Kyber.CPAPKE.Dec}(\mathbf{s}, (\mathbf{u}, v)) \tag{4.9}$$

At this point, reader might find the usage of the parameters $(\mathbf{s}, (\mathbf{u}, v))$ confusing, and wonder why we didn't use the notation $(c := (c_1, c_2), sk)$ instead. The truth is that the parameters $(\mathbf{s}, (\mathbf{u}, v))$ are actually coming from the Kyber.CPAPKE.Dec(c, sk)

algorithm presented in Algorithm 6.

$$\mathbf{u} \leftarrow \text{Decompress}_q(\text{Decode}_{d_u}(c), d_u) \tag{4.10}$$

$$v \leftarrow \text{Decompress}_q(\text{Decode}_{d_v}(c + d_u \cdot k \cdot n/8), dv) \tag{4.11}$$

$$\hat{\mathbf{s}} \leftarrow \text{Decode}_{12}(sk) \tag{4.12}$$

This FO transformation involves the re-encrpypting (Refer to the Line 6) the output of the Kyber.CPAPKE.Dec($\mathbf{s}, (\mathbf{u}, v)$) Algorithm (Refer to the Line 4). In Algorithm 11, a candidate ciphertext c' is generated through running the Kyber.CPA.Enc(pk, m', r') Algorithm (refer to Line 6), and then compared to the public (and input) ciphertext c (refer to Line 7) that we actually started with. Since the Kyber.CPAPKE.Dec() and Kyber.CPAPKE.Enc() algorithms are deterministic, if there is no adversary corrupts the message m, then $c = c'$.

This aims to identify and prevent any maliciously created ciphertexts that might be used to expose the secret key.

Algorithm 10 Kyber.CPAPKE.Dec(c, sk): decryption [1]

Input: Secret key $sk \in B^{12 \cdot k \cdot n/8}$

Input: Ciphertext $c \in B^{d_u \cdot k \cdot n/8 + d_v \cdot n/8}$

Output: Message $m \in B^{32}$

1: $\mathbf{u} \leftarrow \text{Decompress}_q(\text{Decode}_{d_u}(c), d_u)$
2: $v \leftarrow \text{Decompress}_q(\text{Decode}_{d_v}(c + d_u \cdot k \cdot n/8), d_v)$
3: $\hat{\mathbf{s}} \leftarrow \text{Decode}_{12}(sk)$
4: $m \leftarrow \text{Encode}_1(\text{Compress}_q(v - \text{NTT}^{-1}(\hat{\mathbf{s}}^T \circ \text{NTT}(\mathbf{u})), 1))$ ▷
   $m \leftarrow \text{Compress}_q(v - \mathbf{s}^T \mathbf{u}, 1)$
5: **return** m

Below, you will see the Kyber.CCAPKE.Decaps(c, sk) algorithm that is transformed from Kyber.CPAPKE.Dec(c, sk) using the FO transform.

Algorithm 11 Kyber.CCAKEM.Decaps(c, sk): decryption [1]

Input: Ciphertext $c \in B^{d_u \cdot k \cdot n/8 + d_v \cdot n/8}$

Input: Secret key $sk \in B^{24 \cdot k \cdot n/8 + 96}$

Output: Shared ephemeral key $K \in B^*$

1: $pk \leftarrow sk + 12 \cdot k \cdot n/8$
2: $h \leftarrow sk + 24 \cdot k \cdot n/8 + 32 \in B^{32}$
3: $z \leftarrow sk + 24 \cdot k \cdot n/8 + 64$
4: $m' \leftarrow$ Kyber.CPAPKE.Dec$(\mathbf{s}, (\mathbf{u}, v))$
5: $(\overline{K}', r') \leftarrow G(m' || h)$
6: $c' \leftarrow$ Kyber.CPAPKE.Enc(pk, m', r')
7: **if** $c = c'$ **then**
8: $\quad K \leftarrow$ KDF$(\overline{K}' || H(c))$
9: **else**
10: $\quad K \leftarrow$ KDF$(z || H(c))$
11: **end if**
12: **return** K

The essence of CCA security is then manifested in the comparison between c and c'. If they match, the algorithm perceives the ciphertext as valid, deriving the shared key K from the blinded key $\overline{K}'$ and the hash of c. In cases of mismatch, indicative of potential tampering or incorrect generation of the received ciphertext c, the algorithm reverts to a secure fallback mechanism, utilizing the random value z to derive the session key K.

This comparison method reinforces the security by validating the encryption and decryption symmetry of the provided ciphertext c, without exposing any insights into the decryption processes or the internal states, even in instances of discrepancies. The return of a key derived from a random value z in such cases assures the prevention of information leakage, integral for maintaining CCA security.

4.9.1 Explanation of the Length of Certain Algorithmic Components

Recall, in the CCA-secure KeyGen() algorithm, the secret key sk was constructed as follows:

$$sk \leftarrow (sk' \parallel pk \parallel H(pk) \parallel z)$$

Where sk' was derived from the CPA-secure KeyGen() and represented as:

$$sk' \leftarrow \text{Encode}_{12}(\hat{\mathbf{s}} \bmod q)$$

The length of sk' is $\frac{12 \cdot k \cdot n}{8}$ and the cumulative length of sk is as follows:

$$|sk| = (24 \cdot k \cdot n)/8 + 96$$

Consider sk as a book segmented into four chapters:

- Chapter 1: $sk' \in \mathcal{B}^{\frac{12 \cdot k \cdot n}{8}}$
- Chapter 2: $pk \in \mathcal{B}^{\frac{12 \cdot k \cdot n}{8}+32}$
- Chapter 3: $H(pk) \in \mathcal{B}^{32}$
- Chapter 4: $z \in \mathcal{B}^{32}$

Acquiring pk via $pk \leftarrow sk + 12 \cdot k \cdot n/8$ is analogous to opening the book and bypassing the first chapter, pointing directly to the beginning of pk in the concatenated string:

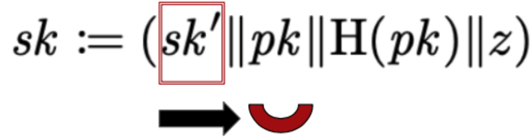


Figure 4.10: Reading the pk from sk

- $pk \leftarrow sk + 12 \cdot k \cdot n/8$

The next steps, represented by reading through chapters, illustrate the extraction of $H(pk) = h$.

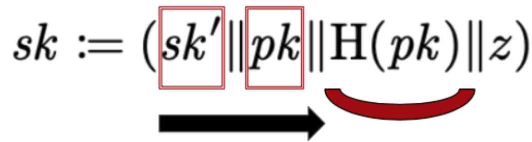


Figure 4.11: Reading the $H(pk)$ from sk

- $h \leftarrow sk + 24 \cdot k \cdot n/8 + 32 \in B^{32}$
  - The length of sk': $(12 \times k \times n)/8$
  - The length of pk: $(12 \times k \times n)/8 + 32$
  - The length of $sk' + pk$: $(24 \times k \times n)/8 + 32$

The z from the composed string sk, highlighting the consecutive composition of the string components.

$$sk := (sk' \| pk \| \mathrm{H}(pk) \| z)$$

Figure 4.12: Reading the z from sk

- $z \leftarrow sk + 24 \cdot k \cdot n/8 + 64$
  - The length of sk': $(12 \times k \times n)/8$
  - The length of pk: $(12 \times k \times n)/8 + 32$
  - The length of $H(pk)$: 32
  - The length of $sk' + pk + H(pk)$: $(24 \times k \times n)/8 + 64$

4.9.2 Leakage Risk Point of the Algorithm

Now, let us turn our focus to the Kyber.CCAKEM.Decaps(c, sk) algorithm to learn the steps that are sensitive and requires to be masked to avoid any side-channel leakage. **On the left**, we have the inputs ciphertext c and secret key s. Note that ciphertext c is the encapsulated secret message m (refer to Line 23). **On the right**, we derive a shared *ephemeral session key* **K**.

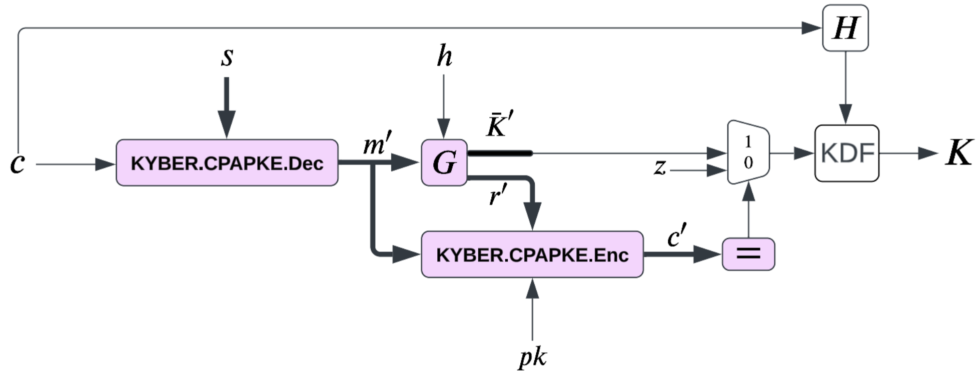


Figure 4.13: Fujisaki-Okamoto Transform for Kyber Key Encapsulation Mechanism (KEM)

Notice that $m' \leftarrow$ Kyber.CPAPKE.Dec$(\mathbf{s}, (\mathbf{u}, v))$ is getting re-encrypted using the *fully-deterministic* encryption algorithm 5. This encryption process results in a ciphertext c'. This ciphertext c' gets compared with the original ciphertext c that we have actually started with.

If these two ciphertexts are equivalent $(c = c')$, then the *Key Derivation Function* (KDF) will leverage the hash function **H** on the **secret key-dependent variable** $\overline{K}'$ as the output of $(\overline{K}', r') \leftarrow G(m'||h)$.

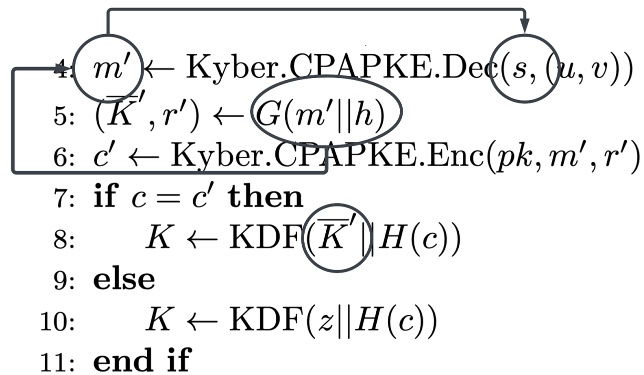


Figure 4.14: $\overline{K}'$ as a Secret Key Dependent Value

If $(c \neq c')$, then a random static number $z \leftarrow sk + 24 \cdot k \cdot n/8 + 64$ is used to drive the key $K \leftarrow$ KDF$(z||H(c))$.

Note that every component or step given in pink in the Figure 4.13 should be masked to some degree. The special case here is that we do not have to mask the $\overline{K}'$[5] at the

[5] Authors put this into different words, "being able to unmask this $\overline{K}'$ [14].

moment we realize that $(c = c')$. In addition, we do not have to mask the comparison result, whether $(c = c')$ as it is either 0 or 1.

The central subject of this thesis is the compression function employed within Kyber.CPAPKE.Dec(). Given that this function directly process the secret key as an input, it presents a substantial risk of being vulnerable to side-channel attacks. Therefore, the very next section delves more intricately into the segments of CRYSTALS-Kyber that necessitate masking. We will discuss the deficiencies in the attention accorded to the masking of the Compress_q function and the inadequacies inherent in existing methodologies given Kyber's choice of prime modulo. Additionally, we will explore how the bit-slided binary search method [14] elegantly addresses the prevailing issues.

CHAPTER 5

DEEP DIVE INTO THE HIGHER-ORDER ONE-BIT COMPRESSION ALGORITHM

In this chapter we will discuss the "**Higher-Order One-Bit Compression**" algorithm proposed by [14]. This chapter does not introduce anything new to the algorithm itself, but rather provides a detailed explanation. The significance of this algorithm lies in its novelty. Even though there have been numerous masked implementations for the functions depicted in Figure 5.1, there were no masked implementations for the **Compress**$_q$ function (the green box in the Fig. 5.1) prior to this study.

5.1 Introduction to the Algorithm

The algorithm that will be the focus of this chapter is shown in Figure 5.1, specifically, the green box (**Compress**$_q$) presented on the right.

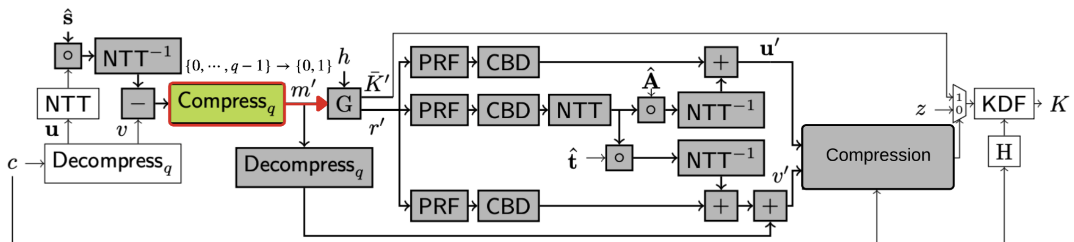


Figure 5.1: Kyber.CCAKEM.Decaps() in Detail, Updated from [14].

Earlier, we provided explicit definitions for the **Compress**$_q$ and **Decompress**$_q$ functions used in CRYSTAL-Kyber (refer to Definition 4.2.5). These functions are applied to a polynomial $x \in \mathbb{R}_q$ and a vector of polynomials $\mathbf{x} \in \mathbb{R}_q^k$, with compression or

decompression operations being applied to each coefficient individually [1]. Every coefficient of a polynomial in Kyber is taken from the coefficient domain $[0, q-1]$. Finally, the reader must understand that when we talk about compressing $\mathbf{x}$, we mean compressing each coefficient of each polynomial in the corresponding array of polynomials into a bit-string.

Algorithm 12 Kyber.CPAPKE.Dec(c, sk): decryption [1]

Input: Secret key $sk \in B^{12 \cdot k \cdot n/8}$

Input: Ciphertext $c \in B^{d_u \cdot k \cdot n/8 + d_v \cdot n/8}$

Output: Message $m \in B^{32}$

1: $\mathbf{u} \leftarrow \text{Decompress}_q(\text{Decode}_{d_u}(c), d_u)$

2: $v \leftarrow \text{Decompress}_q(\text{Decode}_{d_v}(c + d_u \cdot k \cdot n/8), d_v)$

3: $\hat{\mathbf{s}} \leftarrow \text{Decode}_{12}(sk)$

4: $m \leftarrow \text{Encode}_1(\text{Compress}_q(v - \text{NTT}^{-1}(\hat{\mathbf{s}}^T \circ \text{NTT}(\mathbf{u})), 1))$ ▷

Final Result: $m \leftarrow \text{Compress}_q(v - \mathbf{s}^T\mathbf{u}, 1)$

5: **return** m

The **Higher-Order One-Bit Compression** algorithm is designed to compression function that takes place in Kyber.CPAPKE.Dec (refer to Line 4 in Alg. 12). In this process, each coefficient of the resulting polynomial $v - \mathbf{s}^T\mathbf{u}$, falling within the range $[0, q-1]$, is compressed via the Compress_q function into a single bit, either 1 or 0[1]. This approach lends the algorithm its name, 1-bit compression.

However, before we proceed any further, it is imperative to define what *high-order* signifies. Higher-order implies that a single coefficient must be divided into more than two shares (*first*-order masking). Consequently, in implementations utilizing higher-order masking, an attacker would need to access more than one intermediate point (*multiple points*) related to a sensitive value (e.g., a coefficient) to bypass the algorithm's protective measures.

For the sake of simplification, let us say that we have a sensitive value $x \in \mathbb{Z}_q$ such that it has been divided into n_s arithmetical shares (n-tuple) to be processes by various steps (through multiple gadgets) in a cryptographical algorithm. The resulting n_s-

[1] The quantity $v - \mathbf{s}^T\mathbf{u}$ is decrypted to 1 if it is closer to $\lceil q/2 \rfloor$ than to 0. Otherwise, it is decrypted to 0.

tuple can be denoted as $x^{(\cdot)}$ [14].

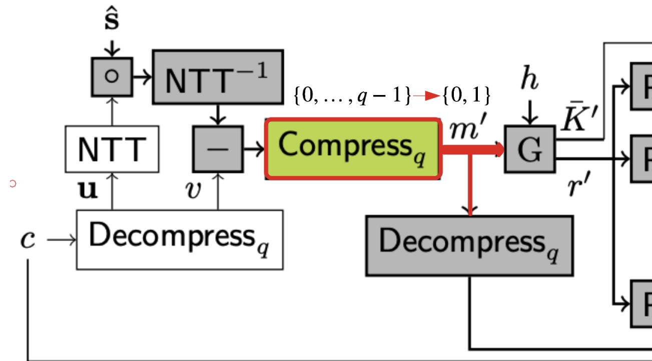
Let us denote x as the arithmetic summation of n_s secrets as follows.

$$x \equiv x^{(0)} + x^{(1)} + x^{(2)} + \cdots + x^{(n_s-1)} \mod q.$$

In that case, if an adversary somehow gains access to t shares such at $t > 1$, in that case it means that adversary can leverage a higher-order attack.

Now, we need to explain why the algorithm is called '1-bit compression'. This is because each coefficient in a polynomial $a \in \mathbb{Z}_{q=3329}[X](X^{256}+1)$, as a polynomial in CRYSTAL-Kyber [13], contains 256 coefficients taken from $\mathbb{Z}_q$ and is compressed into a single bit for the sake of efficiency. In other words, when this algorithm operates on a polynomial a from $\mathbb{R}_q$, the output is a message with a string length of 256 bits, which can be denoted as follows: $m \in \mathbb{Z}_{2^{256}}$. This message, denoted as m', is subsequently re-encrypted by the Kyber.CCAKEM.Decaps() algorithm, generating the ciphertext c', which is then compared with the public ciphertext c to ascertain whether any alterations have occurred during decryption.

As we keep saying "*compression*", it is time for is introduce the *unmasked* compression method used in Kyber.CPAPKE.Dec() algorithm [13].



Arithmetic-to-Boolean Conversion
Figure 5.2: 1-Bit Conversion [14]

5.2 Unmasked Compression in Kyber.CPAPKE.Dec() Algorithm

The primary purpose and rationale of the Compress_q and Decompress_q functions in CRYSTALS-Kyber [13] is to discard some *low-order* bits in ciphertext, minimizing

its size without substantially compromising the security of the scheme (as eliminating low-order bits doesn't notably impact the decryption algorithm's correctness probability) [1].

This is because the most significant bits (MSB) of the polynomial coefficients are the most important bits for determining the correctness of the decryption. The least significant bits, on the other hand, are less important and can be discarded without significantly affecting the security.

In Kyber.CPAPKE.Dec() algorithm [13], the Compress_q function works by taking the **domain of each polynomial coefficient**, which is the set of all integers between 0 and q (i.e, $[0, q-1]$), and splitting it into two disjoint intervals.

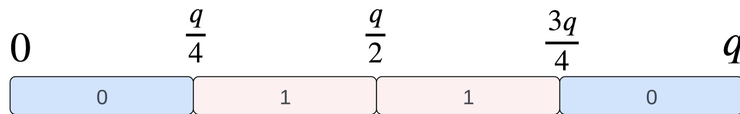


Figure 5.3: Two Disjoint Interval to Compress a Polynomial Coefficient into a Single Bit [15]

The compression function then assigns a value to each coefficient, depending on which interval it falls into. For coefficients in the interval of $[\frac{q}{4}, \frac{3q}{4}]$ (the one coloured in pink in the Figure 5.3), the compression function assigns the value 1. For coefficients in the second interval [0, $\frac{q}{4}$] and [$\frac{3q}{4}, q$] (the ones coloured in blue in the Figure 5.3), the compression function assigns the value 0.

$$\text{Compress}_q(x, 1) = \lceil (2/q) \cdot x \rfloor \mod 2 = \begin{cases} 1, & \text{if } \frac{q}{4} < x < \frac{3q}{4}, \\ 0, & \text{otherwise.} \end{cases} \tag{5.1}$$

The $\text{Compress}_q(x, d = 1)$ takes the element x such that $x \in \mathbb{Z}_q$ and gives an integer in $\{0, \ldots 2^d - 1\}$ as an output such that $d < \lceil log_2(q) \rceil$. Hence, the output of the $\text{Compress}_q(x, 1)$ function will either be 0 or 1.

Notice that a coefficient x is divided by q and rounded to the closest integer. This is quite hard to mask as we do not have a *masked rounding* and *masked division* operations [14]. Hence, it becomes evident that we need to come up with some adaptive and different approaches to mask this function.

5.2.1 Toy Example for Unmasked Compression Function

In this section, we will present a toy example of the unmasked compression function, where $q = 11$. This creates a coefficient domain with numbers ranging from 0 to $q-1$. Using Eq. (5.1), we will demonstrate into which interval each coefficient falls, or in other words, how it is compressed.

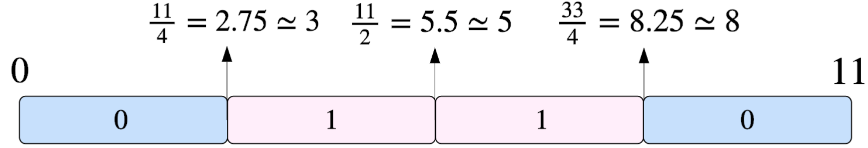


Figure 5.4: Toy Example for Unmasked Compression Function of CRYSTALS-Kyber

.

- $\text{Compress}_q(1, 1) = \lceil \frac{2 \cdot 1}{11} = 0.18 \rfloor = 0 \equiv 0 \mod 2$
- $\text{Compress}_q(2, 1) = \lceil \frac{2 \cdot 2}{11} = 0.36 \rfloor = 0 \equiv 0 \mod 2$
- $\text{Compress}_q(3, 1) = \lceil \frac{2 \cdot 3}{11} = 0.54 \rfloor = 1 \equiv 1 \mod 2$
- $\text{Compress}_q(4, 1) = \lceil \frac{2 \cdot 4}{11} = 0.72 \rfloor = 1 \equiv 1 \mod 2$
- $\text{Compress}_q(5, 1) = \lceil \frac{2 \cdot 5}{11} = \frac{10}{11} = 0.90 \rfloor = 1 \equiv 1 \mod 2$
- $\text{Compress}_q(6, 1) = \lceil \frac{2 \cdot 6}{11} = \frac{12}{11} = 1,09 \rfloor = 1 \equiv 1 \mod 2$
- $\text{Compress}_q(7, 1) = \lceil \frac{2 \cdot 7}{11} = \frac{14}{11} = 1,27 \rfloor = 1 \equiv 1 \mod 2$
- $\text{Compress}_q(8, 1) = \lceil \frac{2 \cdot 8}{11} = \frac{18}{11} = 1,45 \rfloor = 1 \equiv 1 \mod 2$
- $\text{Compress}_q(9, 1) = \lceil \frac{2 \cdot 9}{11} = \frac{18}{11} = 1,63 \rfloor = 2 \equiv 0 \mod 2$
- $\text{Compress}_q(10, 1) = \lceil \frac{2 \cdot 10}{11} = \frac{20}{11} = 1,81 \rfloor = 2 \equiv 0 \mod 2$

5.3 Trivial Masking Approach for Algorithms that Uses Power-of-Two Modulo Like Saber

In the trivial approach for masking with modulo 2^k, the coefficient domain is *shifted* by a certain offset to create a new interval with two equal intervals. This is done by adding $\lfloor \frac{q}{4} \rfloor$ to the polynomial coefficient.

This results in a new shifted compression function that determines whether the coefficient will be 0 or 1:

$$\text{Compress}^s_q(x) := \begin{cases} 0, & \text{if } x < \frac{q}{2}, \\ 1, & \text{otherwise.} \end{cases} \quad (5.2)$$

$$\text{Compress}_q(x, 1) = \text{Compress}^s_q(x + \lfloor \frac{q}{4} \rfloor \mod q) \quad (5.3)$$

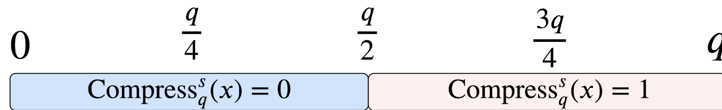


Figure 5.5: Shifting the Polynomial Coefficient with Certain Offset to Mask It [14]

Unlike CRYSTAL-Kyber, many other post quantum (PQC) algorithms like *Saber* uses a modulo q such that $q = 2^k$. And for algorithms like Saber, this naive shifting method can be efficiently implemented.

This is because, the **M**ost **S**ignificant **B**it of the sensitive coefficient x will be a great indicator to tell whether the coefficient will fall into the first half of the interval, MSB$(x) = 0$ in the case where $x < \frac{q}{2}$.

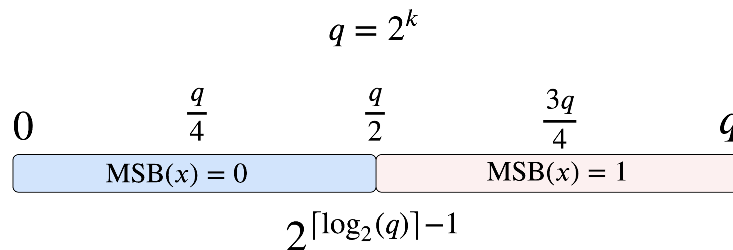


Figure 5.6: Saber ($q = 2^k$), Shifted Compress^s_q Function [14]

Given that modulo $q = 2^k$, its most significant bit (MSB) will have a value of $2^{\lceil log_2(q) \rceil - 1} = 2^{k-1}$. Moreover, $\frac{q}{2} = \frac{2^k}{2} = 2^{k-1}$. As a result, the ability of the interval to be divided into two disjoint intervals of equal spacing by the MSB helps us determine whether a share $x \in \mathbb{Z}_q$ is greater than or equal to $\lfloor \frac{q}{2} \rfloor$.

Unfortunately, in Kyber, where the module is prime $q = 3329$, the interval space is not equally divided by *specific bits* as in Saber. The **MSB** in Kyber has a specific offset (348) from (348) over $\frac{q}{2}$ as it is given in the figure below [15]:

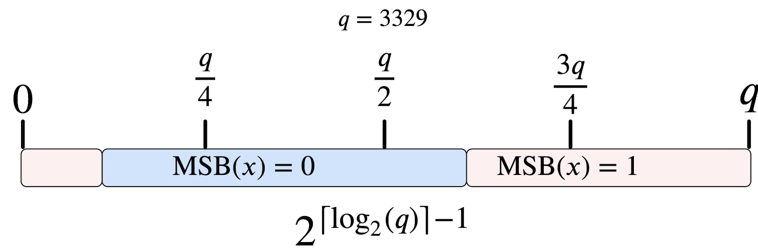


Figure 5.7: Offset Between the MSB and $\frac{q}{2}$ [14]

Thus, the most significant bit (MSB) approach does not work for the Kyber.

The reason for this offset is that CRYSTAL-Kyber uses $q = 3329$. Here, we see that the value of the MSB (2^{11}) corresponds to the "2048" in the coefficient domain.

$$
\begin{aligned}
&= 2^{\lceil log_2(q) \rceil - 1} \\
&= 2^{\lceil log_2(3329) \rceil - 1} \\
&= 2^{\lceil 11.7 \rceil - 1} \\
&= 2^{12-1} \\
&= 2^{11} \\
&= 2048
\end{aligned}
$$

However,

$$
\begin{aligned}
\lfloor q/2 \rfloor &= \\
\lfloor 3329/2 \rfloor &= \\
\lfloor 1664.5 \rfloor &= \\
1664 &
\end{aligned}
$$

So, there is a certain offset value, $2048 - 1664 = 348$. Because the interval is not divided into equal parts as in Saber, the MSB approach cannot be directly applied to CRYSTALS-Kyber and thus necessitates an update.

5.4 Higher-Order One-Bit Compression

As previously emphasized, CRYSTALS-Kyber, in contrast to other PQC schemes like Saber, employs a prime modulo q, where the latter utilizes a 2^k modulo. While the unmasked compression function in Kyber may seem straightforward, introducing masking brings about substantial overhead and integrity challenges, particularly when current methods are applied to mask the compression function.

To overcome this overhead, the authors [14] introduce a novel approach for handling arbitrary modulo q, including both prime and non-prime ones (such as *power-of-two modulo*). This approach offers several advantages over existing methods, particularly when compared to the approach presented in [38]. Notably, the proposed algorithm reduces the *number of conversions required per coefficient to just one*, specifically the Arithmetic-to-Boolean (**A2B**) conversion. The significance of this improvement lies in the efficiency and speed of the algorithm. By minimizing the number of conversions, the computational overhead is reduced, leading to faster execution times.

In this method, the authors has introduced a masking method that leverages a ***bit-sliced binary search algorithm***. In order to implement a ***masked compression function***, authors first take the coefficients of a polynomial that needs to be compressed. For instance, let us say that we have a polynomial $a \in \mathbb{Z}_q[X]$. As I mentioned above, a polynomial in CRYSTAL-Kyber has $(n = 256)$ coefficients where each of these coefficients can be denoted as a_i for $i = \{0, \cdots 255\}$ such that

$$a \equiv a_0 + a_1 X + a_2 X^2 + ... + a_{(255)} * X^{255} \bmod q.$$

In the proposed algorithm [14], authors split a sensitive value (in this case, a polynomial coefficient that needs to be masked) into n_s shares. In other words, a coefficient $a_i \in \mathbb{Z}_q$ is split into n_s-tuple where each share is also in $\mathbb{Z}_q$. These shares can be split arithmetically or in a Boolean format when necessary.

For instance, the $\mathbf{n_s}$**-tuple** arithmetic share representation of a value $a \in \mathbb{Z}_q$ is expressed as

$$a^{(\cdot)_A} = (a^{(0)_A}, \cdots a^{(n_s-1)_A}),$$

which comprises n_s arithmetic shares $a^{(i)_A}$ with $0 \leq i < n_s$. The coefficient a can be

written as follows:

$$a \equiv a^{(0)_A} + a^{(1)_A} + \cdots + a^{(n_s-1)_A} \mod q.$$

In the Boolean case, the n_s-tuple representation of a secret coefficient $a \in \mathbb{Z}_q^2$ can be denoted as

$$a^{(\cdot)_B} = (a^{(0)_B}, \cdots a^{(n_s-1)_B}),$$

which comprises n_s Boolean shares $a^{(i)_B}$ with $0 \leq i < n_s$. This can be written as follows:

$$a = a^{(0)_B} \bigoplus a^{(1)_B} + \bigoplus + \cdots \bigoplus a^{(n_s-1)_B}$$

Now, let's go back to the algorithm [14]. Having splitting it into n_s shares, the algorithm [14] shifts ***each arithmetic share of the secret coefficient*** by a certain offset. (adding an offset $\lfloor \frac{q}{4} = 832 \rfloor$ in mod q to each share of $a_i^{(\cdot)_A}$).

The $\text{Compress}_q^s(x)$ function that shifts each coefficient is given below.

$$\text{Compress}_q^s(x) := \begin{cases} 0, & \text{if } x < \frac{q}{2}, \\ 1, & \text{otherwise.} \end{cases} \tag{5.4}$$

$$\text{Compress}_q(x, 1) = \text{Compress}_q^s(x + \lfloor \frac{q}{4} \rfloor \mod q) \tag{5.5}$$

Next, having each share (n_s) of the secret coefficient shifted by a certain offset, we apply an Arithmetic-to-Boolean (**A2B**) conversion to create 12-bit length n_s Boolean shares for the coefficient. The reason why the conversion output is 12-bit length long is given below.

$$\begin{aligned} k &= \lceil log_2(q) \rceil \\ &= \lceil log_2(3329) \rceil \\ &\approx \lceil 11.7067 \rceil \\ &= 12 \end{aligned}$$

Now that we have the Boolean-shares, the algorithm performs a **Bitslicing** on the n_s-tuple of a Boolean-shared $a_i^{(\cdot)_B}$ bits. In other words, since $a_i^{(\cdot)_B}$ consists of 12-bit length n_s shares, the algorithm slices $a_i^{(\cdot)_B}$ into 12 bits.

$$(a_{11}, a_{10}, a_9, a_8, a_7, a_6, a_5, a_4, a_3, a_2, a_1, a_0)_2.$$

Here, while a_{11} is the most significant bit (MSB), a_0 is the least significant one (LSB). Bitslicing process is done to perform an efficient binary search algorithm.

Below, we provide the exact pseudo-code taken from [14]. However, we would like to address a potential point of confusion in the notation first. From the initial three lines, a meticulous reader may discern that the for-loop is running 256 times, given that a polynomial in $\mathbb{R}_q = \mathbb{Z}_q[X](X^{256}+1)$ has 256 coefficients. The oddity arises next. It seems as if the algorithm performs a shifting operation only on the first share $(a_i^{(0)_A})$ of each coefficient. It's important to clarify that this shifting operation is not limited to the first coefficient share, but is applied to all of them. Additionally, in the main article [14], the authors state that this shifting operation is not performed iteratively, but rather in parallel. In other words, it is carried out simultaneously for each coefficient of a polynomial. The same very thing applies to vectors of polynomials, too.

Algorithm 13 Masked Compress$(x, 1) = \text{Compress}_q^s(x + \lfloor \frac{q}{4} \rfloor \bmod q)$

Input: An arithmetic sharing $a^{(\cdot)_A}$ of a polynomial $a \in \mathbb{Z}_q[X]$.

Output: A Boolean sharing $m'^{(\cdot)_B}$ of the message $m' = \text{Compress}_q(a, 1) \in \mathbb{Z}2^{256}$.

1: **for** $i = 0$ to 255 **do**
2: $\quad a_i^{(0)_A} \leftarrow a_i^{(0)_A} + \lfloor \frac{q}{4} \rfloor \bmod q)$ ▷ G_1
3: $\quad a_i^{(\cdot)_B} \leftarrow \text{Arithmetic-to-Boolean}(a_i^{(\cdot)_A})$ ▷ G_2
4: **end for**
5: $x^{(\cdot)_B} \leftarrow \text{Bitslice}(a^{(\cdot)_B})$ ▷ G_3
6: $m'^{(\cdot)_B} \leftarrow \text{SecAND}(\text{SecREF}(\neg x_8^{(\cdot)_B}), x_7^{(\cdot)_B}))$ ▷ G_4 : NOT, G_5 & G_6
7: $m'^{(\cdot)_B} \leftarrow \text{SecREF}(\text{SecXOR}(m'^{(\cdot)_B}, x_8^{(\cdot)_B}))$ ▷ G_7 : SecXOR, G_8 : SecREF
8: $m'^{(\cdot)_B} \leftarrow \text{SecAND}(m'^{(\cdot)_B}, x_9^{(\cdot)_B})$ ▷ G_9
9: $m'^{(\cdot)_B} \leftarrow \text{SecAND}(m'^{(\cdot)_B}, x_{10}^{(\cdot)_B})$ ▷ G_{10}
10: $m'^{(\cdot)_B} \leftarrow \text{SecAND}(m'^{(\cdot)_B}, \neg x_{11}^{(\cdot)_B})$ ▷ G_{11} : NOT, G_{12} : SecAND
11: $m'^{(\cdot)_B} \leftarrow \text{SecXOR}(m'^{(\cdot)_B}, x_{11}^{(\cdot)_B})$ ▷ G_{13}
12: **return** $m'^{(\cdot)_B}$

Now, we are going to examine the cases that proves why this masked compression function actually works. There are five cases that we are going to study.

5.4.1 The Case 1: Where the First Bit is Set to 1 $(x_{11} = 1)$

In the Figure 5.8, we see the bits that are set and make up the sensitive value x.

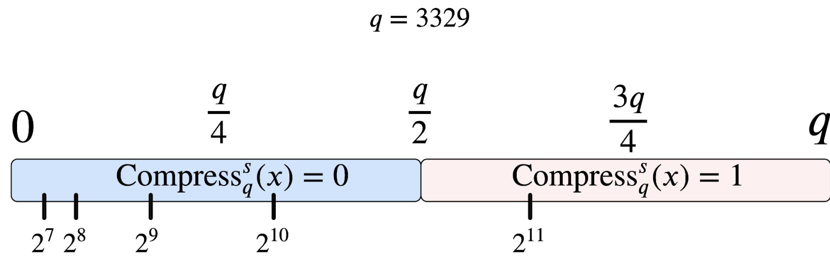


Figure 5.8: Bitslicing the Sensitive Coefficient x [14]

If we see that the eleventh bit is set $(x_{11} = 1)$,

$$2^{\text{MSB}} = 2^{k-1} = 2^{11} = 2048 > \frac{q}{2} \simeq 1664,$$

we directly say that the coefficient will be compressed to 1. In other words, we are sure that the coefficient will be in the particular interval given below:

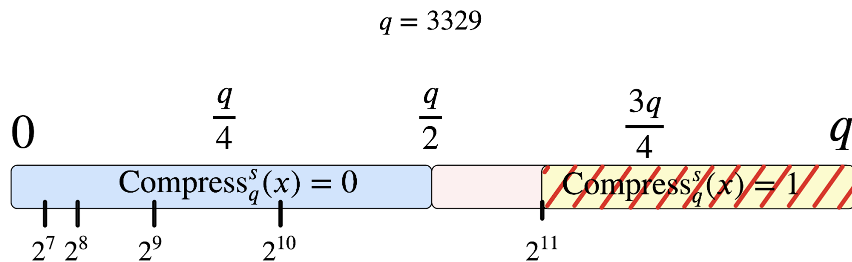
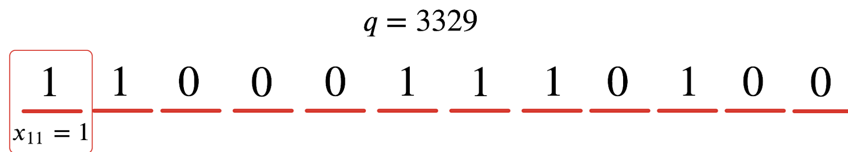


Figure 5.9: When the Most Significant Bit Is Set to 1 [14]



$$(110001110100)_2 = (3188)_{10}$$
$$2^{11} > 1664 \simeq \frac{q}{2}$$
$$(100000000000)_2 = (2048)_{10} > (1664)_{10}$$

Figure 5.10: An example, where a coefficient x is $(110001110100)_2$. This coefficient $x \in [0, q-1]$ is directly compressed to 1 [14].

Hence, first part of our formula starts to build up:

$$\text{Compress}_q^s(x) = x_{11} \oplus (\neg x_{11} \cdot (\ldots))$$

This is because if the 11th bit is set, it means that $x_{11} = 1$. Consequently, "$1 \oplus$ anything" will result in the output of the $\text{Compress}_q^s(x)$ function being 1. Otherwise, if $x_{11} = 0$, we need to examine the other bits. Note how the second part begins by negating the 11th bit to make it 1.

5.4.2 The Case 2: Where $(x_{11} = 0, x_{10} = 0)$

In the second case, the 11th and 10th bits might be set to 0. Notice no matter how rest of the bits are set to 1, if 11th and 10th bit are set to 0,

$$(001111111111)_2 = (1023)_{10} < \frac{q}{2} \simeq 1664,$$

the coefficient is doomed to be compressed to 0.

If the $x_{11} = 0$ but $x_{10} = 1$, we can look for the remaining bits. Hence, the formula for the $\text{Compress}_q^s(x)$ functions continues to build up.

$$\text{Compress}_q^s(x) = x_{11} \oplus (\neg x_{11} \cdot x_{10} \cdot (\ldots))$$

5.4.3 The Case 3: Where $(x_{11} = 0, x_{10} = 1, x_9 = 0)$

In the case of where 11th and 9th bits are not set, but the 10th bit is set, again, the compression is doomed to be compressed to 0. The reason is that even in the best scenario,

$$(010111111111)_2 = (1535)_{10} < \frac{q}{2} \simeq 1664,$$

the value of the coefficient is less then $\frac{q}{2}$.

If 10th and 9th bit are set, then we can continue looking at remaining bits. Hence, the formula for the $\text{Compress}_q^s(x)$ functions continues to build up.

$$\text{Compress}_q^s(x) = x_{11} \oplus (\neg x_{11} \cdot x_{10} \cdot x_9 \cdot (\ldots))$$

The reason why we keep implying the *need for looking at remaining bits* is that even if it is case of $(x_{11} = 0, x_{10} = 1, x_9 = 1)$, as long as the remaining bits are not set accordingly, the coefficient can still be compressed to 0.

For instance,

$$(011000000000)_2 = (1536)_{10} < \frac{q}{2} \simeq 1664,$$

gets compressed to 0.

5.4.4 The Case 4: Where $(x_{11} = 0, x_{10} = 1, x_9 = 1, x_8 = 1)$

In another case, the 11th bit may not be set, but the 10th, 9th, and 8th bits may be set $(x_{11} = 0, x_{10} = 1, x_9 = 1, x_8 = 1)$.

This particular case implies that the coefficient will be approximately compressed to an interval given below, which implies to be 1.

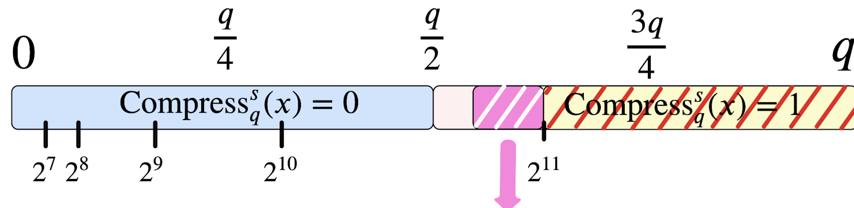


Figure 5.11: An Example Coefficient Compressed to 1

It because, even the worst case, where only the 10th, 9th and 8th bits are set, the value of the share is still bigger than $\frac{q}{2} \simeq 1664$.

$$(011100000000)_2 = (1792)_{10}$$

Thus, the $\text{Compress}_q^s(x)$ becomes the following.

$$\text{Compress}_q^s(x) = x_{11} \oplus (\neg x_{11} \cdot x_{10} \cdot x_9 \cdot (x_8 \oplus (\neg x_8 \cdot (\ldots))))$$

If 8th bit is not set but 10th and 9th bits are set, then we need to look for the 7th bit.

5.4.5 The Case 5: Where $(x_{10} = 1, x_9 = 1, x_8 = 0, x_7 = 1)$

Let's look at another example where the 10th, 9th and 7th bits are set to 1. In this case,

$$2^{10} + 2^9 + 2^7 = 1664 = \lfloor \frac{q}{2} \rfloor,$$

the coefficient x should be compressed to 1.

So, the $\text{Compress}_q^s(x)$ becomes the following.

$$\text{Compress}_q^s(x) = x_{11} \oplus (\neg x_{11} \cdot x_{10} \cdot x_9 \cdot (x_8 \oplus (\neg x_8 \cdot x_7))) \tag{5.6}$$

What we need to understand from these five cases is that when the Most Significant Bit (MSB) is not set to 1, we need to account the *remaining bits*. We repeat this process until each coefficient (of the polynomial being compressed) is mapped to a single bit: {0, 1}.

$$p(X) = a_{255} \cdot X^{255} + a_{254} \cdot X^{254} + \ldots + a_1 \cdot X^1 + a_0 \in Z_{3329}[X]$$

1 or 0 1 or 0 1 or 0 1 or 0

$$m' = \text{Compress}_q(a, 1) \in Z_{2^{256}}$$

Figure 5.12: Compression: Mapping Each Coefficient to a Single Bit. [14]

As the reader might have noticed, in the case of Kyber, where $\lfloor \frac{q}{2} \rfloor = 1664$, only the bits 11th, 10th, 9th, 8th and 7th are taken into consideration. Hence, the Compress_q^s computation is performed as given in Eq. (5.6).

If $x_{11} = 1$, then $2^{11} > 1664$, and the coefficient should be compressed to 1. In this case, the remaining bits are irrelevant, and the function directly maps the coefficient to 1.

If $x_{11} = 0$, we need to consider the remaining bits to determine if the coefficient is greater than or equal to 1664:

- If $x_{10} = 1$ and $x_9 = 1$, then the coefficient is in the range [1536, 2048). Further analysis of x_8 and x_7 is required:

- If $x_8 = 1$, the coefficient is in the range [1664, 2048) and should be compressed to 1, regardless of the value of x_7.
- If $x_8 = 0$ and $x_7 = 1$, the coefficient is exactly 1664, and it should be compressed to 1.

- In all other cases (where $x_{10} = 0, x_9 = 0$), the coefficient is less than 1664, and it should be compressed to 0.

Now let's break down the equation 5.6:

- x_{11}: If the 11th bit is 1, the coefficient is already greater than 1664, and the output is 1.
- $\neg x_{11} \cdot x_{10} \cdot x_9$: This term checks if the first three bits are 011. If true, we need to consider the values of x_8 and x_7 to determine if the coefficient is greater than or equal to 1664.
- $(x_8 \oplus (\neg x_8 \cdot x_7))$: This term evaluates to 1 if either $x_8 = 1$ or $x_8 = 0$ and $x_7 = 1$. Otherwise, it evaluates to 0.

Therefore, the $\text{Compress}(x, 1) = \text{Compress}_q^s(x + \lfloor \frac{q}{4} \rfloor \bmod q)$ function is designed to map the coefficient to 1 if its value is greater than or equal to 1664, and 0 otherwise, using the conditions we analyzed.

To optimize both security and efficiency of a masked implementation of the compression function ($\text{Compress}_q(x, 1)$), it is essential to address several key components.

- First, standard operations should be replaced with their secure counterparts, namely **Sec**XOR and **Sec**AND, to ensure robust protection.
- Second, by transforming Boolean shares of the polynomial into a bit-sliced representation, **parallel computation of all coefficients** becomes possible, enhancing the efficiency based on the target platform's word size. Moreover, the utilization of t-SNI secure conversion and t-NI computation for masked bitwise operations, such as A2B, SecAND, and SecREF, further bolsters security. (In the upcoming sections, we will introduce the t-NI and t-SNI security notions and will prove that the Algorithm 13 is in fact t-SNI secure.)

- The adaptability of the masked compression algorithm (13) allows for the integration of various algorithms that fulfill the required properties, such as those proposed in ([23], [5], and [52]). Additionally, the implementation accounts for platform-specific capabilities, using sequences of bitshift, bitwise OR, and bitwise AND operations to rearrange bits share by share, making it highly versatile and suitable for different platforms.

5.5 Probing Security of the Algorithm 13

As stressed before, masking is a *countermeasure* technique used to mitigate side-channel attacks by breaking down a sensitive variable into multiple shares. In the context of *"Side-Channel Notation and Notions"* presented in (Bos et al., 2021, p. 5) [14], a sensitive variable $x \in \mathbb{Z}_q$ is divided into n_s shares, denoted as $x \equiv x^{(0)_A} + x^{(1)_A} + \cdots + x^{(n_s-1)_A} \mod q$ (referred to as arithmetic shares). These shares collectively form the n_s-tuple $x^{(\cdot)_A}$.

The purpose of this sharing is to enable secure processing of the sensitive variable by the *underlying circuit*. (We will present the underlying circuit for the Algorithm 13 in this section. See the Figure 5.13). By representing x as shares, computations involving x can be performed on the individual shares, *reducing the exposure of the sensitive information.*

To ascertain the *security of these shared implementations* in a comprehensive manner, we need to refer to the **t-probing model**, a crucial theoretical construct introduced in 2003 [27].

The t-probing model in cryptography represents an adversary with access to up to t intermediate variables in the system. This model is essential for evaluating the security of masked circuits against side-channel attacks, where the goal is to ensure that *any combination of* t variables within the circuit does not reveal information about the secret. The objective is to protect the secret data even when the attacker can probe t variables.

Proving the security of individual functions or *gadgets* against probing attacks is a

necessary first step. However, when considering compositions of multiple *gadgets*, especially at **higher orders (t > 1)**, the security analysis becomes more challenging. In higher-order scenarios, an attacker can obtain *information about more than one intermediate value at a time*, requiring a comprehensive evaluation of the combined effects of multiple gadgets or functions within the cryptographic algorithm. As the order t increases, the attacker becomes more powerful, necessitating more complex security analyses.

To address the security of constructions involving multiple gadgets and higher-order attacks, authors [14] have relied on the principles of t-Non-Interference (t-NI) and t-Strong-Non-Interference (t-SNI).

5.5.1 Gadgets: The Fundamental Units in Cryptographic Systems

Before discussing t-probing security, we need to define a fundamental term: the **gadget**. Within the scope of masking and side-channel analysis, a gadget refers to a basic building block or a small sub-circuit in a larger cryptographic system. Each gadget is an individual function or operation (e.g., addition, multiplication, or bitwise operations) that accomplishes a specific task. The combined workings of these gadgets form the complete cryptographic circuit.

5.5.2 Gadgets Employed in the Higher-Order One-Bit Compression Algorithm 13

In this subsection, we will list the gadgets used in the Algorithm 13. Note that the Figure 5.13 is directly taken from the [14].

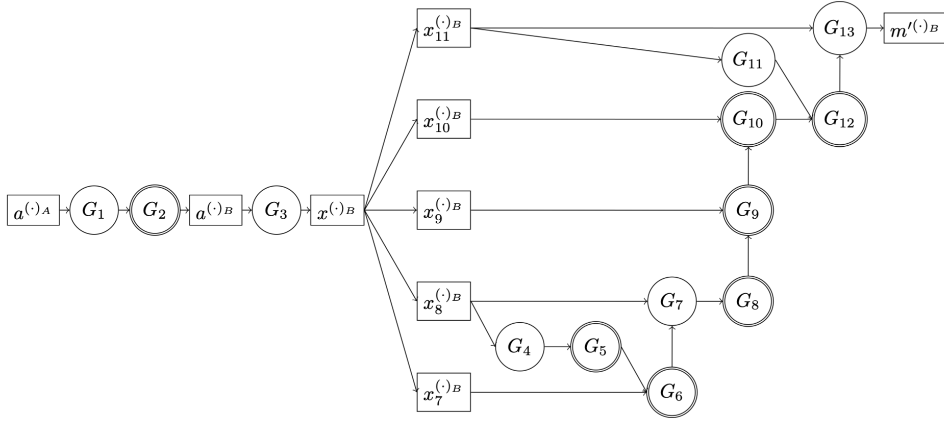


Figure 5.13: Gadgets in the Algorithm 13 in [14].

- G_1 (t-NI)[2]: Refers to the *subtraction operation* in the Line 2. The gadget G_1 subtracts a certain offset value $\lfloor \frac{q}{4} \rfloor \mod q$ from each share of $a^{(i)_A}$. Remember this is where the actual shifting (*masking*) operation happens.

- G_2 (t-SNI)[3]: Refers to the *Arithmetic-to-Boolean* (A2B) conversion in the Line 3. This gadget converts the arithmetic shares (there are n_s of them) $a_i^{(\cdot)_A}$ into Boolean shares $a_i^{(\cdot)_B}$ using the A2B conversion. In arithmetical format, $a_i^{(\cdot)_A}$ can be shown as the following,

$$a_i \equiv a_i^{(0)_A} + a_i^{(1)_A} + a_i^{(2)_A} + \cdots + a_i^{(n_s-1)_A} \mod q$$
$$a_i^{(\cdot)_A} = (a^{(0)}, a^{(1)}, a^{(2)}, \cdots, a^{(n_s-1)})$$

  where each arithmetical share is in $\mathbb{Z}_{2^k}$. In the Boolean format, $a_i^{(\cdot)_B}$ can be shown as the following:

$$a_i^{(\cdot)_B} = a_i^{(0)_A} \oplus a_i^{(1)_A} \oplus a_i^{(2)_A} \oplus \cdots \oplus a_i^{(n_s-1)_A}$$
$$a_i^{(\cdot)_B} = (a^{(0)}, a^{(1)}, a^{(2)}, \cdots, a^{(n_s-1)})$$

  where each Boolean share is in $\mathbb{Z}_{2^k}$

- G_3 (NI): Refers to the *Bitslicing* process in the Line 5. This gadget slices the Boolean shares $a_i^{(\cdot)_B}$, each consisting of 12 bits, into 12 separate bits.

- G_4 (NI): Refers to the "*not*" operation in the Line 6. This gadget performs the logical negation (NOT) operation on the shares of $x_8^{(\cdot)_B}$.

[2] NI AND SNI notions will be introduce in this section.
[3] Note that the gadgets that hold the t-SNI property are shown with a bold perimeter in the Figure 5.13

- G_5 (SNI): Refers to the **Sec**REF in the Line 6. This gadget refreshes the secret shares of $x_7^{(\cdot)_B}$ to prevent information leakage.

- G_6 (SNI): Refers to the **Sec**AND in the Line 6. This gadget applies the secret sharing operation (SecAND) to the shares of $x_7^{(\cdot)_B}$ and the refreshed shares of $x_8^{(\cdot)_B}$.

- G_7 (NI): Refers to the Sec**XOR** in the Line 7. This gadget performs the secret sharing operation (SecXOR) on the shares of $m'^{(\cdot)_B}$ and $x_8^{(\cdot)_B}$.

- G_8 (SNI): Refers to the Sec**REF** in Line 7. This gadget refreshes the secret shares of $m'^{(\cdot)_B}$ to protect against information leakage.

- G_9 (SNI): Refers to the Sec**AND** in Line 8. This gadget applies the secret sharing operation (SecAND) to the shares of $m'^{(\cdot)_B}$ and $x_9^{(\cdot)_B}$.

- G_{10} (SNI): Refers to the Sec**AND** in Line 9. This gadget applies the secret sharing operation (SecAND) to the shares of $m'^{(\cdot)B}$ and $x_{10}^{(\cdot)_B}$.

- G_{11} (NI): Refers to the "*not*" in Line 10. This gadget performs the logical negation (NOT) operation on the shares of $x_{11}^{(\cdot)_B}$.

- G_{12} (SNI): Refers to the Sec**AND** in Line 10. This gadget applies the secret sharing operation (SecAND) to the shares of $m'^{(\cdot)B}$ and the negated shares of $x_{11}^{(\cdot)_B}$.

- G_{13} (NI): Refers to the Sec**XOR** in Line 10. This gadget performs the secret sharing operation (SecXOR) on the shares of $m'^{(\cdot)_B}$ and the shares of the result of the previous operation.

5.5.3 t-SNI Security of the Algorithm 13

In the article (Bos et al., 2021), *Theorem 1* establishes the t-SNI security property of Algorithm 13 with $n_s = t + 1$.

In the case where $a^{(\cdot)_A}$ represents the input and $m'^{(\cdot)_B}$ represents the output of Algorithm 13, Theorem 1 states that for any given set of t_{A_1} intermediate variables and any subset $O \subset [0, n_s - 1]$ of the output variables, where the sum of the cardinality of O

and t_{A_1} is less than the total number of shares n_s (i.e., $t_{A_1} + |O| < n_s$), there exists a subset $I \subset [0, n_s - 1]$ of input shares. This subset I, with a cardinality no greater than t_{A_1} (i.e., $|I| \leq t_{A_1}$), enables the perfect simulation of the t_{A_1} intermediate variables and the output variables $m'^{(O)_B}$ using the corresponding input shares $a^{(I)_A}$.

In other words, by considering a carefully chosen subset of the input shares ($I \leq t_{A_1}$), it is possible to perfectly emulate the behavior of the intermediate variables and the desired output, ensuring that the algorithm's execution remains secure against probing attacks.

5.5.4 Proof of the Theorem 1 [14]

Disclaimer: The linear operations in the gadgets G_1, G_4, G_5 and G_{11} are taken as t-NI secure gadgets.

In order to prove the t-SNI security of the Algorithm 13, we need to demonstrate that probes on intermediate and output variables can be perfectly simulated through a subset of intermediate values.

While t_{G_i} denotes the number of internal probe that an adversary can place internally for gadget G_i, the o_{G_i}, as one can predict, denote the output probes for a gadget G_i.

- $t_{A_{13}}$ is the number of probes that an adversary can gain,
- $|O|$ refers to the output shares from the Algorithm 13 [14].

For instance, for the gadget G_{13}, the output probe (output of this particular gadget) can be denoted as $o_{G_{13}}$.

$$t_{A_{13}} = \sum_{i=1}^{13} t_{G_i} + \sum_{i=1}^{12} o_{G_i}, \ |O| = o_{G_{13}}.$$

The main objective of this proof is to show that both internal probes and output shares can be simulated with less than or equal to $t_{A_{13}}$ number of probes.

Simulating G_{13}:

Let's dive a bit deeper into the process of simulating the $t_{G_{13}}$ intermediate and $o_{G_{13}}$ output probes of the t-NI gadget G_{13} [4]

$$m'^{(\cdot)_B} \leftarrow \text{SecAND}(m'^{(\cdot)_B}, \neg x_{11}^{(\cdot)_B}) \qquad \text{G}_{11} : \text{NOT}, \text{G}_{12} : \text{SecAND}$$
$$m'^{(\cdot)_B} \leftarrow \text{SecXOR}(m'^{(\cdot)_B}, x_{11}^{(\cdot)_B}) \qquad \text{G}_{13} : \text{SecXOR}$$

Here, the two inputs to the gadget G_{13} are given as follows:

- $x_{11}^{(\cdot)_B}$: One of the input shares.
- The Output of the Gadget G_{12}: This comes from the **Sec**AND operation in the Line 10 of the Algorithm 13, which is applied to the shares of $m'^{(\cdot)B}$ and the negated shares of $x_{11}^{(\cdot)_B}$.

Hence, when we talk about requiring $t_{G_{13}} + o_{G_{13}}$ shares of these two inputs, it means we need an equivalent amount of information or "share" of these inputs to accurately simulate the probes of the G_{13} gadget [14].

Simulating G_{12}:

In the larger context of an algorithm, composed of multiple t-SNI gadgets like G_{12}, the objective is to minimize the propagation of probes and limit the adversary's information access. This process of limiting information access forms the core of t-SNI (Threshold-Probing Security with Non-Interference) security, and the method of achieving this is through accurate simulation.

In the case of G_{12}, a t-SNI gadget, its internal probes $t_{G_{12}}$ and output probes $o_{G_{12}}$ can be perfectly simulated using only $t_{G_{12}}$ shares of the outputs of gadgets G_{10} and G_{11}. This is possible because of the property of the t-SNI gadgets, which allows the simulation of their internal and output behavior based only on a restricted number of input shares [14].

This is significant because it means the simulation of the gadget doesn't require

[4] A probe in this context refers to information that can be obtained from the gadget during the simulation.

knowledge of all the output shares ($o_{G_{12}}$), which are usually assumed to be accessible to an adversary.

Simulating $G_4 - G_{13}$**:** It is important to note that the notation $t_{x_i^{(\cdot)_B}}$ is used to denote the total shares required to accurately simulate a particular bit $x_i^{(\cdot)_B}$.

To emulate the intermediary and final outputs of the devices G_4 through G_{13}, we necessitate the subsequent quantity of shares for the variables $x_7^{(\cdot)_B}$ through to $x_{11}^{(\cdot)_B}$:

- $\mathbf{t_{x_7^{(\cdot)_B}} = t_{G_6}}$

We need t_{G_6} shares for simulation. This is because G_6 is a SecAND gadget that operates on the shares of $x_7^{(\cdot)_B}$. The output of G_6 is also $x_7^{(\cdot)_B}$ and as G_6 is an SNI gadget, it needs t_{G_6} internal probes to be simulated correctly.

- $\mathbf{t_{x_8^{(\cdot)_B}} = t_{G_4} + o_{G_4} + t_{G_5} + t_{G_7} + o_{G_7} + t_{G_8}}$

$x_8^{(\cdot)_B}$ goes through several gadgets (G_4, G_5, G_7, G_8), each of which may have internal probes (t) or output shares (o).

- G_4 is a "not" operation, which logically negates $x_8^{(\cdot)_B}$. The t_{G_4} and o_{G_4} represent the internal probes and output shares for this operation.
- G_5 is a refresh operation, denoted as t-SNI refresh. The refresh operation is needed to prevent potential information leakage that could happen due to duplications in the number of shares or probes (like the case where you have t_{G_6} shares of both $x_7^{(\cdot)_B}$ and $x_8^{(\cdot)_B}$). The t_{G_5} represents the internal probes for the refresh operation.
- G_7 is a secret XOR operation, denoted as SecXOR. This operation computes the exclusive or of $x_8^{(\cdot)_B}$ with another secret bit $m'^{(\cdot)_B}$. The t_{G_7} and o_{G_7} are the internal probes and output shares for this operation.
- G_8 is another refresh operation. Similar to G_5, it is used to avoid potential information leakage from the SecXOR operation. The term t_{G_8} corresponds to the internal probes for this refresh operation.

In total, to simulate the process of $x_8^{(\cdot)_B}$ correctly through these gadgets, we would need the sum of all these internal probes and output shares, which is why $t_{x_8^{(\cdot)_B}} = t_{G_4} + o_{G_4} + t_{G_5} + t_{G_7} + o_{G_7} + t_{G_8}$. This calculation ensures that the simulation doesn't exceed the secure information handling capacity of these operations, preventing any potential information leak that could compromise the protocol's security.

- $\mathbf{t_{x_9^{(\cdot)B}} = t_{G_9}}$

To simulate G_9, which is another SecAND operation but this time performed on $x_9^{(\cdot)_B}$, you need $t_{x_9^{(\cdot)_B}}$ shares of $x_9^{(\cdot)_B}$.

- $\mathbf{t_{x_{10}^{(\cdot)B}} = t_{G_{10}}}$:

The simulation of this bit requires the total shares from the gadget G_{10}, which is a SecAND gadget. Hence, the total number of shares needed for simulation will be the number of internal probes of G_{10}, denoted as $t_{G_{10}}$. So, we have $t_{x_{10}^{(\cdot)_B}} = t_{G_{10}}$.

- $\mathbf{t_{x_{11}^{(\cdot)B}} = t_{G_{11}}}$:

The simulation of this bit is a bit more complex, as it goes through a series of operations, namely, the "not" operation (G_{11}), SecAND (G_{12}), and SecXOR (G_{13}). Each of these operations would require a certain number of shares for their correct simulation. Hence, the total shares needed for simulating $x_{11}^{(\cdot)_B}$ would be the sum of the internal and output probes from these gadgets. Therefore, we have $t_{x_{11}^{(\cdot)_B}} = t_{G_{11}} + o_{G_{11}} + t_{G_{12}} + t_{G_{13}} + o_{G_{11}}$.

For Bitslice, we add up the total number of shares for each bit to get the total shares $t_{x^{(\cdot)_B}}$. The simulation can only be performed if there are no duplicate entries in the sum, otherwise the simulation would require twice the number of shares for certain bits, which is not feasible. This is why the refresh in G_5 is needed, without which, the simulation would require t_{G_6} shares of both $x_7^{(\cdot)_B}$ and $x_8^{(\cdot)_B}$, effectively needing twice the shares.

The same reasoning applies for G_6 and G_9, where the inputs need to be refreshed to

prevent duplicate shares. However, for other SecAND gadgets, this issue does not occur, and hence, there is no need to refresh their inputs.

The t-NI property of Bitslice allows us to simulate the $t_{x^{(\cdot)_B}}$ shares of $x^{(\cdot)_B}$ with the corresponding number of shares of $a^{(\cdot)_B}$.

Lastly, following the flow through gadgets G_2 and G_1, the simulation of Algorithm 1 requires $|I| = t_{G_1} + o_{G_1} + t_{G_2}$ of the input shares $a^{(\cdot)_A}$. The G_2, with its t-SNI property, allows us to simulate the shares of $a^{(\cdot)_B}$ with only t_{G_2} of its input. This stops the propagation of $t_{x^{(\cdot)_B}}$ to $|I|$, making the simulation more efficient.

CHAPTER 6

ALTERNATIVE MASKING

6.1 Introduction

In this chapter, we will introduce two alternative methods to perform a masked compression for CRYSTALS-Kyber.

- The Double and Check Compression Function (Refer to Section 6.2)
- The Look-Up-Table (LUT) Based Compression Algorithm (Refer to Section 6.3)

Subsequently, we will transition to Section 6.4, in which we will identify a suitable prime number, specifically $q = 7681$, and demonstrate that inspecting the five most significant bits of a number within their corresponding domains $[0, q-1]$ is sufficient. The choice of a prime number is pivotal, as it directly impacts the effectiveness and security of the algorithm, thereby necessitating a thorough examination of its properties and characteristics.

In the end, we'll lay out the corresponding compression functions and go into detail about the number of bit-wise operations the algorithm needs. This exploration will enable a comprehensive understanding of the computational complexity inherent in the algorithm, offering insights into its efficiency and operational demands. By noting the number of bit-wise operations, we can discern the algorithm's resource consumption, allowing for an informed evaluation of its practicality and viability in various applications.

6.2 The Double and Check Compression Method

The Double and Check algorithm provides a method to handle arithmetic shares, introducing a level of *obfuscation*. In this approach, everything up to and including the shifting procedure aligns with Alg. 13. Each coefficient of a polynomial in $\mathbb{Z}_q[X]$ is divided into several "shares", with the property that the sum of the shares modulo q equals the original value. Each share is then shifted by adding $\lfloor q/4 \rfloor$, which makes the interval symmetric around $\lfloor q/2 \rfloor$.

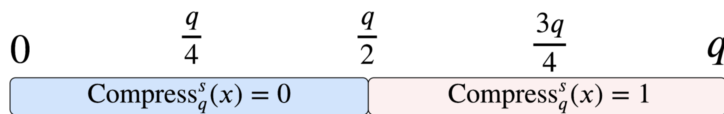


Figure 6.1: Shifting the Polynomial Coefficient with Certain Offset to Mask It [14]

Retaining the symmetric interval property of the coefficient from Alg. 13 ensures consistent overflow behavior beyond $q = 3329$. This allows the algorithm to yield a Boolean output, indicating if the original share exceeded $\lfloor q/2 \rfloor$.

After the symmetry is achieved, the algorithm doubles each shifted share. This doubling acts as an obfuscated method for indirectly gauging if the shifted share, when multiplied by two, surpasses the modulo q. If it does, it implies that the original share was greater than $\lfloor q/2 \rfloor$. This overflow, or surpassing of the modulo, is detected by checking if the doubled share after taking modulo q is different from the raw doubled share (before taking modulo). If these two values are not the same, an overflow has occurred.

Hence, the "Check" in "Double and Check 14" is the procedure of observing whether this overflow took place. When overflow is detected, the algorithm outputs "1", indicating the original share was above the midpoint $\lfloor q/2 \rfloor$. If no overflow occurred, the output is "0".

This method of doubling and checking, rather than making a direct comparison to the midpoint, provides an obfuscated yet effective way to ascertain the position of the original share relative to $\lfloor q/2 \rfloor$. This is instrumental in preserving the integrity and security of the polynomial coefficient shares while still enabling efficient computa-

tion.

Below is a detailed breakdown of the Double and Check Alg. 14. It's crucial to emphasize that the steps outlined are approached in the *simplest manner possible*, without any *parallelization*. For clarity's sake, we employ for-loops to elucidate the algorithm's mechanics.

- Iterate over each coefficient of the polynomial, indexed by i. Given that a polynomial in CRYSTALS-Kyber comprises 256 coefficients, the loop runs from $i = 0$ to $i = 255$.
- For each coefficient, loop over each of its shares (indexed by k).
- For each share, add $\lfloor q/4 \rfloor$ modulo q. This "shifts" the share to make the domain symmetric around $\lfloor q/2 \rfloor$.
- Double the shifted share, assign it to a variable called "*raw_double_share*."
- Take the modulo q of the *raw_double_share*, assign the value to a variable called "*double_share*."
- If *double_share* does not equal the *raw_double_share*, this means there was an overflow over q, which in turn means the share must have been greater than $\lfloor q/2 \rfloor$. This information is stored in a Boolean form. In other words, the result will be either *1* or *0*.
- The algorithm outputs a vector, such as "[1, 0, 1, 1]," when the number of shares for a coefficient is set at 4. Subsequent components of the algorithm, like masked encryption, will use this vector as an input.

6.2.1 Pseudo-code of the Double and Check Algorithm

The Double and Check algorithm (Refer to Alg. 14), introduces an obfuscated method to evaluate polynomial coefficients. Rather than directly comparing to $\lfloor \frac{q}{2} \rfloor$, this approach employs a doubling operation to determine if a shifted share exceeds $\lfloor \frac{q}{2} \rfloor$. While this method preserves integrity and promotes computational efficiency, it does necessitate *modular addition*, which might be more computationally intensive than a

Algorithm 14 Double and Check algorithm

Input: An arithmetic sharing $a^{(\cdot)_A}$ of a polynomial $a \in \mathbb{Z}_q[X]$, with each coefficient is masked by n_s shares such that $a_i^{(\cdot)_A} = a_i^{(0)_A} + a_i^{(1)_A} + \cdots + a_i^{(n_s-1)_A} \mod q$.

Output: A Boolean sharing $m'^{(\cdot)_B}$ indicating if each share is greater than $q/2$ with $m' = \text{Compress}_q(a, 1) \in \mathbb{Z}_{2^{256}}$.

```
1:  for i = 0 to 255 do
2:      for k = 0 to n_s − 1 do
3:          a_i^(k)_A ← a_i^(k)_A + ⌊q/4⌋ mod q
4:          raw_double_share ← a_i^(k)_A + a_i^(k)_A
5:          double_share ← raw_double_share mod q
6:          if double_share = raw_double_share then
7:              m_i^(k)_B ← 0
8:          else
9:              m_i^(k)_B ← 1
10:         end if
11:     end for
12: end for
13: return m^(·)_B
```

bit-wise operation. Nevertheless, this approach offers a naive, obfuscated alternative for coefficient evaluation. Detailed steps of this algorithm are clearly presented in the pseudo-code of Alg. 14.

6.2.2 Toy Example for the Double and Check Algorithm

To better exemplify the algorithm, let us consider a simple example with $n_s = 4$ shares and a modulus $q = 3329$. For clarity, we will focus solely on the coefficient 2419 of the polynomial

$$p(x) = 2419 \cdot x^{255} + 2289 \cdot x^{254} + \cdots + 1723 \cdot x + 3187 \in \mathbb{Z}_{3329}[X].$$

Given the arithmetic shares $[678, 1162, 2406, 1502]$, their summation modulo 3329 yields 2419, consistent with our polynomial coefficient.

To achieve symmetry around $\lfloor q/2 \rfloor = 1664$, we adjust each share by adding $\lfloor q/4 \rfloor = 832$. This results in the updated shares: $[1510, 1994, 3238, 2334]$.

The Double and Check algorithm (Refer to 14) then proceeds by doubling each of these offset shares and reducing mod q. The final step is to check if doubling the offset share overflows q. This is done by comparing if the doubled offset share equals 2 times the original offset share. We calculate the doubled shares modulo 3329 as follows:

- Double $1510 \mod 3329 \equiv 3020$. As this is less than ($q = 3329$), it does not overflow q when doubling, and thus the result for this share is **0**.
- For the other shares $(1994, 3238, 2334)$, when doubled, they all exceed 3329, so the doubling overflows q. This means the original offset shares were greater than $\lfloor q/2 \rfloor = 1664$, and thus the results for these shares are all **1**.
- Result of whether offset shares are greater than 1664: [0, 1, 1, 1]

Essentially, the Double and Check algorithm is providing a mask to reveal whether each share, when shifted by a certain offset and then performed modular addition, is larger than $\lfloor q/2 \rfloor$. If the double of the offset share isn't equal to 2 times the original offset share when reduced modulo q, it means the value has overflowed q, which can only happen if the original offset share is greater than $\lfloor q/2 \rfloor$.

6.2.3 Final Notes On the Double and Check Method

The "Double and Check" method, as illustrated, is predicated on the self-addition of an arithmetic share, followed by a comparison between the doubled value, with and without the application of the modulo operation, to discern whether an overflow has transpired. From a computational standpoint, the following are reflections on this method:

- **Polynomial-Time Addition Operation**

Adding a share to itself ($raw_double_share \leftarrow a_i^{(k)_A} + a_i^{(k)_A}$), we are essentially performing an addition operation. Addition of two numbers (or in this case, adding a number to itself) is a fundamental arithmetic operation that can be done in polynomial time, specifically, it has a linear time complexity, $\mathcal{O}(n)$ is the number of bits in the binary representation of the numbers.

When the Double and Check algorithm performs the addition operation to calculate raw_double_share, it is inherently secure in the sense that it does not expose the system to side-channel leakage risks, due to the polynomial-time nature of the addition operation. Operations that take polynomial time, especially those with lower-degree polynomials, are generally quick and do not allow for substantial variance in execution time, which could potentially be exploited to gather information about the system's state or the data being processed.

In contrast, if the operation were to have a higher time complexity, especially with significant variability, it might create discernible patterns in terms of execution time, power consumption, etc., that could potentially be exploited for side-channel attacks.

To conclude, the polynomial-time addition operation in the Double and Check algorithm is efficient and does not expose the system to side-channel leakage risks due to its consistent and fast execution, thus contributing to the overall security of the algorithm in a computational context.

- **Modulo Reduction**

The operation $double_share \leftarrow raw_double_share \bmod q$ does incorporate a minimal computational overhead, but it is typically considered negligible, especially with modern computational resources.

When the modulus is a prime number ($q = 3329$) and not a power of two (2^k), the operation may not be optimized to a simple bit-wise operation and may thus incur a bit more computational cost, but given the indispensability of the operation in maintaining numerical ranges and preventing overflow, the benefits derived from its application significantly outweigh the minor overhead introduced, preserving the integrity and manageability of the computational process.

- **Comparison Operation**

The comparison within the given algorithm, specifically

$$double_share = raw_double_share,$$

is inherently computationally inexpensive, serving as a mere bitwise comparison.

---

if double_share = raw_double_share **then**
  $m_i^{(k)_B} \leftarrow 0$
else
  $m_i^{(k)_B} \leftarrow 1$
end if

The complexity associated with such a comparison is deemed negligible, owing to its simplistic nature and execution efficiency. A bit-wise comparison operates at the level of individual bits, making it one of the most fundamental and swift operations in computer science.

Furthermore, the merit of this comparison extends to its security attributes; given that it does not hinge on the comparison of secret or sensitive values, there is no requisite for implementing masking, thus mitigating potential vulnerabilities associated with the exposure of sensitive data.

In essence, the minimal computational expense and the non-reliance on sensitive values imbue the algorithm with both operational efficiency and an enhanced level of security.

6.3 Look-Up-Table (LUT) Based Compression Algorithm (32 Entities)

In this section, we are going to introduce a Look-Up-Table (LUT) integration, which represents a classic trade-off between time (computational speed) and space (memory/storage). In many scenarios, especially in cryptography, where speed is paramount, this trade-off is acceptable.

CRYSTALS-Kyber uses a prime modulo $q = 3329$. The masked $\text{Compress}_q^s(x)$ function introduced by [14] first splits polynomial coefficients into random arithmetical shares. Then, to divide the coefficient domain into to symmetric interval (symmetric to $\lfloor \frac{q}{2} \rfloor$), it shifts each share by $\lfloor \frac{q}{4} \rfloor$. This concept is illustrated in Fig. 5.5.

Then, Alg. 1 in [14] checks each share whether it is bigger then $\lfloor \frac{q}{2} \rfloor$, or not. If it is, assigns to 1, otherwise, assigns to 0.

$$
\begin{aligned}
\lfloor \frac{q}{2} \rfloor &= \lfloor \frac{3329}{2} \rfloor \\
&= \lfloor 1664.5 \rfloor \\
&= 1664 \\
&= 2^{10} + 2^9 + 2^7 \\
&= (011010000000)_2
\end{aligned}
$$

Hence, to check whether a 12-bit number in the $[0, q-1]$ domain is bigger then 1664, it is enough for the algorithm to check the *5 most significant bit* of the number that needs to be compared against 1664.

$$
\begin{aligned}
k &= \lceil log_2(q) \rceil \\
&= \lceil log_2(3329) \rceil \\
&= \lceil 11.7067 \rceil \\
&= 12
\end{aligned}
$$

Then, the Alg. 1 [14] runs the following dedicated function for each share of every coefficient.

$$
\text{Compress}_q^s(x) = x_{11} \oplus (\neg x_{11} * x_{10} * x_9 * (x_8 \oplus (\neg x_8 * x_7)))
$$

Notice, how looking at the first 5 most significant bits is enough to decide whether a number in $[0, q-1]$ is bigger than 1664, or not. Thus, instead of doing this bit-wise operations for each and every share within the algorithm, we can leverage a Look-up-Table (LUT) of at most $2^5 = 32$ entities.

In this technique, we uphold the fundamental principle of masking: *splitting a sensitive value $x \in \mathbb{Z}_q$ into smaller shares $(x_0, x_1, \cdots x_{n_s-1})$, and performing the operations on these shares*. The reason behind this is stressed in the work [7]. Carrying

out the operations of an algorithm in the domain defined by the *masking* effectively blocks any potential information leakage associated with the variable x, as it is not directly handled at any point. Rather, the only detectable leakage in the side-channel measurements originates from calculations that involve either x_1 or x_2 [7]. As both arithmetical and Boolean shares of a coefficient x are defined randomly for each run, leakage caused by x_1 and x_2 does not reveal anything about the coefficient x to an attacker. This is why we wanted to preserve the masking nature of the Compress_q^s algorithm, even while integrating a LUT.

6.3.1 Pseudo-code of the Look-Up-Table (LUT) Based Compression Algorithm

To provide clarity on the implementation of the Look-up-Table (LUT) based masked compression, we present Alg. 15. The algorithm takes a secret coefficient $x \in \mathbb{Z}_{3329}[X]$ and splits it into its n_s shares, shifts each share by $\lfloor \frac{q}{4} \rfloor = 832$, and casts each share into its 12-bit Boolean representation to decide if the share is greater than or equal to 1664. This decision is made using a Look-Up-Table.

The table might look like the following, but to have a more detailed look, refer to Table A.3.2).

Table 6.1: Bit Representation, Index, and Values

5-bit	Index	Value
00000	0	0
00001	1	0
...		
01100	12	0
01101	13	1
...		
11110	30	1
11111	31	1

The algorithm requires only 32 entities because we compare a number in the coefficient domain $[0, 3329 - 1]$ to determine if it's greater than or equal to $1664_{10} = (011010000000)_2$. Thus, it's sufficient to look at the 5 most significant bits of a number when comparing it to 1664. As previously mentioned, Alg. 15 integrates

Algorithm 15 LUT-based Compression Algorithm

Input: An arithmetic sharing $a^{(\cdot)_A}$ of a polynomial $a \in \mathbb{Z}_q[X]$, with each coefficient is masked by n_s shares such that $a_i^{(\cdot)_A} = a_i^{(0)_A} + a_i^{(1)_A} + \cdots + a_i^{(n_s-1)_A} \mod q$.

Output: A Boolean sharing $m'^{(\cdot)_B}$ indicating if each offset share is greater than $q/2$.

1: Pre-compute LUT with indices from 0 to 31 where each **index** is a binary representation of an integer i with bits $x_{11}, x_{10}, x_9, x_8, x_7$ and LUT[i] is set to x_{11} XOR (NOT x_{11} * x_{10} * x_9 * (x_8 XOR (NOT x_8 * x_7))) as introduced by [14] in the Algorithm 13.

2: **for** $i = 0$ to 255 **do**

3: **for** $k = 0$ to $n_s - 1$ **do**

4: $a_i^{(k)_A} \leftarrow a_i^{(k)_A} + \lfloor \frac{q}{4} \rfloor \mod q$

5: $a_i^{(k)_B} \leftarrow \text{A2B}(a_i^{(k)_A})$

6: Extract a 5-bit index from the most significant bits of $a_i^{(k)_B}$

7: $m_i^{(k)_B} \leftarrow$ LUT[index] ▷ Use the LUT to determine if the offset share is greater than $\lfloor q/2 \rfloor$

8: **end for**

9: **end for**

10: **return** $m^{(\cdot)_B}$

a pre-computed table of 32 entities (refer to Table A.3.2). Each entity is indexed by the decimal value of its Boolean representation. To generate the LUT as quickly as possible, we used the bit-sliced binary search method proposed by [14]. For details on how the table can be generated with Python, refer to A.3.1.

To exemplify this: assume we have a Boolean representation (011110111001). How does the algorithm leverages the LUT to decide whether it is bigger then or equal to 1664. To determine whether this is greater than or equal to 1664 using the LUT, the algorithm first takes the 5 most significant bits, which are (01111). It then determines that the index is 15. At index 11, it observes a value of 1. Consequently, Algorithm 15 concludes that the number is greater than or equal to 1664.

6.3.2 Toy Example for the Look-Up-Table (LUT) Based Offset and Check Algorithm

To provide a more solid understanding, we implemented a trivial Python implementation for a toy example, which integrates the LUT.

Let's start by understanding the `lut_based_comparison` function. (Refer to Code A.3.3, the lines $26 - 31$). The main goal of this function is to determine, for each compressed share, whether its value (based on the first 5 most significant bits) is greater than or equal to 1664. Instead of conducting a full comparison with the threshold value (1664), the function efficiently uses the LUT. Each entry in the LUT encapsulates the outcome for specific 5-bit patterns, allowing for a faster decision-making process.

Here is the detailed explanation of the function:

- The function initializes an empty list called results.
- For each bits (which refers to each compressed share) in the shares list, the function performs the following steps:
  - Extracts the 5 most significant bits of the current compressed share to form a 5-bit index (Refer to Line 29 in A.3.3).
  - Using this 5-bit index, it fetches the corresponding value from the LUT and appends this value to the results list.
- Once all shares have been processed, the function returns the results list.

Let's take an example:

Assume we have shares = [2035, 725, 287, 1282] (these are shares that have been calculated previously) and a given lookup table lut (as we've provided).

- For the sake of efficiency, we focus our energy on the first share: 2035, convert it into binary, it becomes $(01111111011)_2$.
- Next, we extract the 5 most significant bits: 01111 (or 15 in decimal).

- We then check the value in lut at index 15 which is 1.
- So, the corresponding Boolean value for the share 2035 is 1.
- The function repeats these steps for every shifted share in the list and returns a list of Boolean values.

Finally, the `lut_based_comparison`([2035, 725, 287, 1282], LUT) call would return $[1, 0, 0, 0]$, indicating which shares are greater than $\lfloor \frac{q}{2} \rfloor$. The function is efficient because it uses the top 5 bits to directly look up the result in a pre-computed table.

6.3.3 Notes On the Look-Up-Table-based Approach

The LUT-based approach is more efficient than the Double-and-Check method because it replaces multiple bitwise operations (including an inversion, multiplications, and an XOR operation) with a single lookup operation. The lookup operation has a constant time complexity, meaning it takes the same amount of time regardless of the input size, which is generally faster than performing multiple bitwise operations.

There are some potential concerns or limitations with this approach:

- **Pre-computation Overhead**

The generation of the Look-Up-Table, despite being a one-time procedure, does require computation resources. Although it's relatively negligible given the fixed size of the LUT, this overhead might be taken into account, especially in contexts with constrained computational resources.

- **Reduced Flexibility**

The Look-Up-Table method is tailor-made for the particular problem at hand with a specific q. If any alterations to the problem parameters or related problems are to be addressed, the Look-Up-Table and the corresponding parts of the algorithm would likely need to be restructured.

- **Memory Consumption**

The storage of the Look-Up-Table, albeit not significant due to its small size, will incur some memory overhead.

In summary, while the Look-Up-Table-based approach affords greater computational efficiency, these potential caveats and considerations should be evaluated in deciding whether to adopt this approach.

6.4 Potential Prime Numbers for Non LUT-Based Compression Functions

As we studied earlier, in their work [14], (Bos et al., 2021, p. 7) proposed the $\text{Compress}_s^q(x)$ function. This function only checks the 5 most significant bits (MSB) of a number from the $[0, q-1]$ domain, where $q = 3329$ and using the $\lfloor \frac{q}{2} \rfloor = 1664$ and $(011010000000)_2$. Since only the 5 MSB of the 1664 are set, Alg. 13 can perform a bit-sliced binary search, only examining the 5 most significant bits of a number in a coefficient domain.

This led us to consider whether there exist prime numbers for which examining half of the prime, denoted as $\lfloor \frac{q}{2} \rfloor$, would allow us to only check the first 4 or 3 most significant bits (MSB) of a number to determine if it is greater than or equal to $\lfloor \frac{q}{2} \rfloor$. However, we were unable to find such prime numbers that have a bit length of 11, 12, or 13.

Therefore, we redirected our efforts towards finding prime numbers for which half would yield a value that requires only the inspection of the 5 most significant bits (MSB) of a number in the coefficient domain to determine if it is greater than or equal to $\lfloor \frac{q}{2} \rfloor$.

6.4.1 Masked Compress Functions for Potential Prime Numbers

In this section, we delve into an exploration of potential compression functions for CRYSTAL-Kyber, not strictly within the context where the prime number q is fixed at

Table 6.2: Analysis for the Prime Number 1153

Attribute	**Value**
Number	1153
k	11
$\lfloor \frac{q}{2} \rfloor$	$576.5 \simeq 576$
Binary representation of 576	$(01001000000)_2$
$\text{Compress}_q^s(x)$	$x_{10} \oplus (\neg x_{10} \cdot x_9 \cdot x_8 \cdot x_7 \cdot (x_6 \oplus (\neg x_6 \cdot x_5)))$
Number of XOR operations	2
Number of AND operations	5
Number of NEG operations	2

3329. Instead, we broaden our perspective to consider a diverse set of prime numbers:

$$\{1153, 1409, 7681\}$$

Our goal is to uncover and understand the array of compression functions that may arise and be suitable under varying prime number circumstances within this specified set.

Table 6.2 delineates the compress function formulated for the prime number 1153, spotlighting the binary representation of 576, which is half of 1153. The articulated $\text{Compress}_q^s(x)$ function manifests a moderate level of intricacy, primarily characterized by the prevalence of AND operations. The quantity of XOR and NEG operations fall within conventional bounds, contributing to the overall complexity.

Nevertheless, despite the congruent levels of complexity, the prime number q retains a relatively diminutive magnitude. Consequently, this results in the polynomial coefficients also being comparably smaller. This phenomenon potentially jeopardizes the security of the algorithm, especially considering that the s and e vectors are derived through central binomial distributions. The smaller coefficients could inadvertently introduce vulnerabilities or reduce the robustness of the cryptographic processes, emphasizing the crucial balance between computational efficiency and security integrity in algorithm design and implementation.

Table 6.3 illustrates a seemingly more computationally efficient compression function for the prime 1409, with a diminished count of AND operations compared to the method outlined in [14]. This reduced complexity could potentially be advantageous in scenarios where operational efficiency is paramount. However, it's pivotal to note

that the utilization of a smaller prime number, such as 1409, inherently incurs security vulnerabilities. To date, there is a conspicuous absence of implementations utilizing such a low prime, indicating that the possible computational advantages are likely overshadowed by the resultant security compromises.

Table 6.3: Analysis for the Prime Number 1409

Attribute	**Value**
Number	1409
k	11
$\lfloor \frac{q}{2} \rfloor$	$704.5 \simeq 704$
Binary representation of 704	$(01011000000)_2$
$\text{Compress}_q^s(x)$	$x_{10} \oplus (\neg x_{10} \cdot x_9 \cdot (x_8 \oplus (\neg x_8 \cdot x_7)))$
Number of XOR operations	2
Number of AND operations	3
Number of NEG operations	2

Lastly, for the prime 7681, Table 6.4 elucidates its compress function derived from the binary design of 3840.

Given the prime $q = 7681$, which is the smallest prime fulfilling $1 \equiv q \mod 2n$ for $n = 256$, it was originally suggested by CCA secure CRYSTALS-Kyber [4] and exemplifies efficiency in Table 6.4. This prime ensures swift NTT multiplication, a pivotal property for lattice-based cryptographic schemes, where polynomial multiplication is notably computation-intensive. The characteristics of $q = 7681$ make it an exemplary candidate, allowing for a compression function with fewer XOR and NEG operations compared to the method proposed by [14]. This results in an enhancement in operational speed and effectiveness, balancing computational efficiency and security needs. The specific selection of q is crucial for leveraging the benefits of the fast NTT multiplication property, and it facilitates the existence of a $2n$-th primitive root of unity in the finite field, aligning the computational requisites and algorithmic executions for optimal cryptographic robustness.

Building upon the foundational work by [14], we introduce a modified masked compression function, depicted in Alg. 16, adapted to operate with $q = 7681$. This enhanced function enables an efficient and secure transformation of an arithmetic sharing of a polynomial $a \in \mathbb{Z}_q[X]$ into a Boolean sharing of the compressed message

Table 6.4: Analysis for the Prime Number 7681

Attribute	**Value**
Number	7681
k	13
$\lfloor \frac{q}{2} \rfloor$	3840.5 $\simeq$ 3840
Binary representation of 3840	(0111100000000)2
Compress$^s_q(x)$	$x_{13} \oplus (\neg x_{13} \cdot x_{12} \cdot x_{11} \cdot x_{10} \cdot x_9)$
Number of XOR operations	1
Number of AND operations	4
Number of NEG operations	1

Algorithm 16 Modified Masked Compress$(x, 1)$

Input: An arithmetic sharing $a^{(\cdot)_A}$ of a polynomial $a \in \mathbb{Z}_q[X]$.

Output: A Boolean sharing $m'^{(\cdot)_B}$ of the message $m' = \text{Compress}_q(a, 1) \in \mathbb{Z}_{2^{256}}$.

1: **for** $i = 0$ to 255 **do**
2: $\quad a_i^{(0)_A} \leftarrow a_i^{(0)_A} + \lfloor \frac{q}{4} \rfloor \bmod q$
3: $\quad a_i^{(\cdot)_B} \leftarrow$ Arithmetic-to-Boolean$(a_i^{(\cdot)_A})$
4: **end for**
5: $x^{(\cdot)_B} \leftarrow$ Bitslice$(a^{(\cdot)_B})$
6: $m'^{(\cdot)_B} \leftarrow$ SecAND$(x_9^{(\cdot)_B}, x_{10}^{(\cdot)_B})$
7: $m'^{(\cdot)_B} \leftarrow$ SecAND$(m'^{(\cdot)_B}, x_{11}^{(\cdot)_B})$
8: $m'^{(\cdot)_B} \leftarrow$ SecAND$(m'^{(\cdot)_B}, x_{12}^{(\cdot)_B})$
9: $m'^{(\cdot)_B} \leftarrow$ SecAND$(m'^{(\cdot)_B}, \neg x_{13}^{(\cdot)_B})$
10: $m'^{(\cdot)_B} \leftarrow$ SecXOR$(m'^{(\cdot)_B}, x_{13}^{(\cdot)_B})$
11: **return** $m'^{(\cdot)_B}$

$m' = \text{Compress}_q(a, 1) \in \mathbb{Z}_{2^{256}}$. The selection of $q = 7681$ is pivotal in this context as it is the smallest prime conforming to $1 \equiv q \mod 2n$ for $n = 256$, fostering swift NTT multiplication, a crucial attribute in lattice-based cryptographic schemes. This modification aims to optimally balance computational overhead and cryptographic resilience, while retaining the essence of the masked compression function proposed by [14].

CHAPTER 7

CONCLUSION

In conclusion, our research has delved into formulating techniques aimed at mitigating the risk of side-channel leakage of sensitive polynomial coefficients during the compression step in Kyber.CPAPKE.Dec() Algorithm. In this thesis, we've introduced two innovative strategies: the "Double and Check" algorithm and the LUT-based method.

The "Double and Check" algorithm aspires to uphold data integrity and offers computational efficiency, being grounded in polynomial-time addition. This approach, entailing conditional checks, seems promising as it doesn't process the secret data directly, potentially minimizing side-channel vulnerabilities. While it manifests a straightforward interaction with data, it might, theoretically, consume more resources in comparison to the bit-wise operations explicated in [14]. Consequently, exhaustive research and practical implementation are requisite to validate its efficacy and to perform a comparative analysis. On the other hand, the LUT-based approach, grounded on modular arithmetic and supported by pre-computed tables, aims to strike a balance between data protection and computational efficiency. Its premise is straightforward, leveraging tables to facilitate quicker operations. However, the simplicity also introduces a dilemma—memory utilization. Even though the tables are compact, they might present challenges when integrated into settings with limited memory, particularly some hardware contexts.

As side-channel attack techniques become increasingly sophisticated, our research is not a definitive answer but rather a simple suggestion. The techniques we've introduced here are initial proposals. Each holds potential but also demands further

research. They serve as contributions that, with further refinement and testing, could play a role in the broader narrative of side-channel risk mitigation within CRYSTALS-Kyber. We hope that this discourse stimulates further research and validation, ensuring that any implementation is both informed and resistant.

REFERENCES

[1] R. Avanzi, J. Bos, L. Ducas, E. Kiltz, T. Lepoint, V. Lyubashevsky, J. M. Schanck, P. Schwabe, G. Seiler, and D. Stehlé, Crystals-kyber algorithm specifications and supporting documentation, NIST PQC Round, 2(4), pp. 1–43, 2019.

[2] G. Barthe, S. Belaïd, F. Dupressoir, P.-A. Fouque, B. Grégoire, and P.-Y. Strub, Verified proofs of higher-order masking, in *Advances in Cryptology–EUROCRYPT 2015: 34th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Sofia, Bulgaria, April 26-30, 2015, Proceedings, Part I*, pp. 457–485, Springer, 2015.

[3] G. Barthe, S. Belaïd, F. Dupressoir, P.-A. Fouque, B. Grégoire, P.-Y. Strub, and R. Zucchini, Strong non-interference and type-directed higher-order masking, in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pp. 116–129, 2016.

[4] G. Barthe, S. Belaïd, T. Espitau, P.-A. Fouque, B. Grégoire, M. Rossi, and M. Tibouchi, Masking the glp lattice-based signature scheme at any order, in *Advances in Cryptology–EUROCRYPT 2018: 37th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Tel Aviv, Israel, April 29-May 3, 2018 Proceedings, Part II 37*, pp. 354–384, Springer, 2018.

[5] A. Battistello, J.-S. Coron, E. Prouff, and R. Zeitoun, Horizontal side-channel attacks and countermeasures on the isw masking scheme, in *Cryptographic Hardware and Embedded Systems–CHES 2016: 18th International Conference, Santa Barbara, CA, USA, August 17-19, 2016, Proceedings*, pp. 23–39, Springer, 2016.

[6] M. V. Beirendonck, J.-P. D'Anvers, A. Karmakar, J. Balasch, and I. Verbauwhede, A side-channel resistant implementation of saber, Cryptology ePrint Archive, Paper 2020/733, 2020, `https://eprint.iacr.org/2020/733`.

[7] M. V. Beirendonck, J.-P. D'anvers, A. Karmakar, J. Balasch, and I. Verbauwhede, A side-channel-resistant implementation of saber, ACM Journal on Emerging Technologies in Computing Systems (JETC), 17(2), pp. 1–26, 2021.

[8] S. Belaïd and T. Güneysu, *Smart Card Research and Advanced Applications: 18th International Conference, CARDIS 2019, Prague, Czech Republic, Novem-*

ber 11–13, 2019, Revised Selected Papers, volume 11833, Springer Nature, 2020.

[9] M. Bellare and P. Rogaway, The security of triple encryption and a framework for code-based game-playing proofs, in *Advances in Cryptology-EUROCRYPT 2006: 24th Annual International Conference on the Theory and Applications of Cryptographic Techniques, St. Petersburg, Russia, May 28-June 1, 2006. Proceedings 25*, pp. 409–426, Springer, 2006.

[10] L. Bettale, J.-S. Coron, and R. Zeitoun, Improved high-order conversion from boolean to arithmetic masking, IACR Transactions on Cryptographic Hardware and Embedded Systems, pp. 22–45, 2018.

[11] S. Bhasin, J.-P. D'Anvers, D. Heinz, T. Pöppelmann, and M. Van Beirendonck, Attacking and defending masked polynomial comparison for lattice-based cryptography, IACR Transactions on Cryptographic Hardware and Embedded Systems, pp. 334–359, 2021.

[12] A. Blum, M. Furst, M. Kearns, and R. J. Lipton, Cryptographic primitives based on hard learning problems, in *Annual International Cryptology Conference*, pp. 278–291, Springer, 1993.

[13] J. Bos, L. Ducas, E. Kiltz, T. Lepoint, V. Lyubashevsky, J. M. Schanck, P. Schwabe, G. Seiler, and D. Stehlé, Crystals-kyber: a cca-secure module-lattice-based kem, in *2018 IEEE European Symposium on Security and Privacy (EuroS&P)*, pp. 353–367, IEEE, 2018.

[14] J. W. Bos, M. Gourjon, J. Renes, T. Schneider, and C. Van Vredendaal, Masking kyber: First-and higher-order implementations, IACR Transactions on Cryptographic Hardware and Embedded Systems, pp. 173–214, 2021.

[15] J. W. Bos, M. Gourjon, J. Renes, T. Schneider, and C. van Vredendaal, Masking kyber: First- and higher-order implementations, YouTube video in TheIACR Conference Recording, 2023, [Online; accessed 28-June-2023].

[16] Z. Brakerski, C. Gentry, and V. Vaikuntanathan, (leveled) fully homomorphic encryption without bootstrapping, ACM Transactions on Computation Theory (TOCT), 6(3), pp. 1–36, 2014.

[17] O. Bronchain and G. Cassiers, Bitslicing arithmetic/boolean masking conversions for fun and profit: with application to lattice-based kems, IACR Transactions on Cryptographic Hardware and Embedded Systems, pp. 553–588, 2022.

[18] S. Chari, C. S. Jutla, J. R. Rao, and P. Rohatgi, Towards sound approaches to counteract power-analysis attacks, in *Advances in Cryptology—CRYPTO'99: 19th Annual International Cryptology Conference Santa Barbara, California, USA, August 15–19, 1999 Proceedings 19*, pp. 398–412, Springer, 1999.

[19] E. Dubrova, K. Ngo, J. Gärtner, and R. Wang, Breaking a fifth-order masked implementation of crystals-kyber by copy-paste, in *Proceedings of the 10th ACM Asia Public-Key Cryptography Workshop*, pp. 10–20, 2023.

[20] T. Fritzmann, M. Van Beirendonck, D. B. Roy, P. Karl, T. Schamberger, I. Verbauwhede, and G. Sigl, Masked accelerators and instruction set extensions for post-quantum cryptography, Cryptology ePrint Archive, 2021.

[21] E. Fujisaki and T. Okamoto, Secure integration of asymmetric and symmetric encryption schemes, in *Advances in Cryptology—CRYPTO'99: 19th Annual International Cryptology Conference Santa Barbara, California, USA, August 15–19, 1999 Proceedings*, pp. 537–554, Springer, 1999.

[22] S. Goldwasser and D. Micciancio, Complexity of lattice problems: a cryptographic perspective, 2002.

[23] D. Goudarzi, A. Journault, M. Rivain, and F.-X. Standaert, Secure multiplication for bitslice higher-order masking: Optimisation and comparison, in *Constructive Side-Channel Analysis and Secure Design: 9th International Workshop, COSADE 2018, Singapore, April 23–24, 2018, Proceedings*, pp. 3–22, Springer, 2018.

[24] H. Groß, S. Mangard, and T. Korak, An efficient side-channel protected aes implementation with arbitrary protection order, in *Cryptographers' Track at the RSA Conference*, pp. 95–112, Springer, 2017.

[25] D. Heinz, M. J. Kannwischer, G. Land, T. Pöppelmann, P. Schwabe, and D. Sprenkels, First-order masked kyber on arm cortex-m4, Cryptology ePrint Archive, 2022.

[26] D. Hofheinz, K. Hövelmanns, and E. Kiltz, A modular analysis of the fujisaki-okamoto transformation, in *Theory of Cryptography: 15th International Conference, TCC 2017, Baltimore, MD, USA, November 12-15, 2017, Proceedings, Part I*, pp. 341–371, Springer, 2017.

[27] Y. Ishai, A. Sahai, and D. Wagner, Private circuits: Securing hardware against probing attacks, in *Advances in Cryptology-CRYPTO 2003: 23rd Annual International Cryptology Conference, Santa Barbara, California, USA, August 17-21, 2003. Proceedings 23*, pp. 463–481, Springer, 2003.

[28] Y. Ji, R. Wang, K. Ngo, E. Dubrova, and L. Backlund, A side-channel attack on a hardware implementation of crystals-kyber, in *2023 IEEE European Test Symposium (ETS)*, pp. 1–5, IEEE, 2023.

[29] P. C. Kocher, Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems, in *Advances in Cryptology—CRYPTO'96: 16th Annual International Cryptology Conference Santa Barbara, California, USA August 18–22, 1996 Proceedings 16*, pp. 104–113, Springer, 1996.

[30] A. Langlois and D. Stehlé, Worst-case to average-case reductions for module lattices, Designs, Codes and Cryptography, 75(3), pp. 565–599, 2015.

[31] V. Lyubashevsky, C. Peikert, and O. Regev, On ideal lattices and learning with errors over rings, Journal of the ACM (JACM), 60(6), pp. 1–35, 2013.

[32] T. S. Messerges, Securing the aes finalists against power analysis attacks, in *International Workshop on Fast Software Encryption*, pp. 150–164, Springer, 2000.

[33] D. Micciancio and O. Regev, Lattice-based cryptography, Post-quantum cryptography, pp. 147–191, 2009.

[34] V. Migliore, B. Gérard, M. Tibouchi, and P.-A. Fouque, Masking dilithium: Efficient implementation and side-channel evaluation, in *Applied Cryptography and Network Security: 17th International Conference, ACNS 2019, Bogota, Colombia, June 5–7, 2019, Proceedings 17*, pp. 344–362, Springer, 2019.

[35] National Institute of Standards and Technology (NIST), Status report on the third round of the nist post-quantum cryptography standardization process, Technical Report NIST IR 8413, National Institute of Standards and Technology, 2022.

[36] K. Ngo, E. Dubrova, Q. Guo, and T. Johansson, A side-channel attack on a masked ind-cca secure saber kem implementation, IACR Transactions on Cryptographic Hardware and Embedded Systems, pp. 676–707, 2021.

[37] M. B. Niasar, Lattices and kyber pqc presentation, YouTube, 2022, [Online; accessed 28-June-2023].

[38] T. Oder, T. Schneider, T. Pöppelmann, and T. Güneysu, Practical cca2-secure and masked ring-lwe implementation, Cryptology ePrint Archive, 123(4), pp. 101–120, 2016.

[39] C. Peikert, Public-key cryptosystems from the worst-case shortest vector problem, in *Proceedings of the forty-first annual ACM symposium on Theory of computing*, pp. 333–342, 2009.

[40] C. Peikert, Lattice cryptography for the internet, in *Post-Quantum Cryptography: 6th International Workshop, PQCrypto 2014, Waterloo, ON, Canada, October 1-3, 2014. Proceedings 6*, pp. 197–219, Springer, 2014.

[41] C. Peikert, EECS 598: Lattices in Cryptography, 2015, university of Michigan, Course notes for EECS 598.

[42] C. Peikert, The learning with errors problem and cryptographic applications, in *Lattices: Algorithms, Complexity, and Cryptography Boot Camp*, The Simons Institute for the Theory of Computing, Calvin Lab Auditorium, Jan 2020, no abstract available.

[43] C. Peikert, O. Regev, and N. Stephens-Davidowitz, Pseudorandomness of ring-lwe for any ring and modulus, in *Proceedings of the 49th Annual ACM SIGACT Symposium on Theory of Computing*, pp. 461–473, 2017.

[44] C. Peikert et al., A decade of lattice cryptography, Foundations and Trends® in Theoretical Computer Science, 10(4), pp. 283–424, 2016.

[45] J. Proos and C. Zalka, Shor's discrete logarithm quantum algorithm for elliptic curves, arXiv preprint quant-ph/0301141, 2003.

[46] E. Prouff and M. Rivain, Masking against side-channel attacks: A formal security proof, in *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pp. 142–159, Springer, 2013.

[47] C. Rackoff and D. R. Simon, Non-interactive zero-knowledge proof of knowledge and chosen ciphertext attack, in *Annual international cryptology conference*, pp. 433–444, Springer, 1991.

[48] O. Regev, 0368.4282: Lattices in Computer Science, Course notes for 0368.4282, 2009, school of Computer Science, Tel Aviv University.

[49] O. Regev, On lattices, learning with errors, random linear codes, and cryptography, Journal of the ACM (JACM), 56(6), pp. 1–40, 2009.

[50] O. Regev, The learning with errors problem (invited survey), in *2010 IEEE 25th Annual Conference on Computational Complexity*, pp. 191–204, IEEE, 2010.

[51] O. Reparaz, S. Sinha Roy, F. Vercauteren, and I. Verbauwhede, A masked ring-lwe implementation, in *International Workshop on Cryptographic Hardware and Embedded Systems*, pp. 683–702, Springer, 2015.

[52] T. Schneider, C. Paglialonga, T. Oder, and T. Güneysu, Efficiently masking binomial sampling at arbitrary orders for lattice-based crypto, in *Public-Key Cryptography–PKC 2019: 22nd IACR International Conference on Practice and Theory of Public-Key Cryptography, Beijing, China, April 14-17, 2019, Proceedings, Part II 22*, pp. 534–564, Springer, 2019.

[53] P. W. Shor, Algorithms for quantum computation: discrete logarithms and factoring, in *Proceedings 35th annual symposium on foundations of computer science*, pp. 124–134, Ieee, 1994.

[54] V. Shoup, Sequences of games: a tool for taming complexity in security proofs, cryptology eprint archive, 2004.

[55] J. H. Silverman and W. Whyte, Timing attacks on ntruencrypt via variation in the number of hash calls, in *Topics in Cryptology–CT-RSA 2007: The Cryptographers' Track at the RSA Conference 2007, San Francisco, CA, USA, February 5-9, 2007. Proceedings*, pp. 208–224, Springer, 2006.

[56] D. Stebila, Post-quantum cryptography from the learning with errors problem, in *2021 IEEE North American School of Information Theory*, University of Waterloo, 2021.

[57] D. Stehlé, R. Steinfeld, K. Tanaka, and K. Xagawa, Efficient public key encryption based on ideal lattices, in *Advances in Cryptology–ASIACRYPT 2009: 15th International Conference on the Theory and Application of Cryptology and Information Security, Tokyo, Japan, December 6-10, 2009. Proceedings 15*, pp. 617–635, Springer, 2009.

[58] N. Stephens-Davidowitz, Search-to-decision reductions for lattice problems with approximation factors (slightly) greater than one, arXiv preprint arXiv:1512.04138, 2015.

[59] R. Ueno, K. Xagawa, Y. Tanaka, A. Ito, J. Takahashi, and N. Homma, Curse of re-encryption: A generic power/em analysis on post-quantum kems, IACR Transactions on Cryptographic Hardware and Embedded Systems, pp. 296–322, 2022.

[60] R. Wang, M. Brisfors, and E. Dubrova, A side-channel attack on a bitsliced higher-order masked crystals-kyber implementation, Cryptology ePrint Archive, 2023.

[61] Z. Xu, O. Pemberton, S. S. Roy, D. Oswald, W. Yao, and Z. Zheng, Magnifying side-channel leakage of lattice-based cryptosystems with chosen ciphertexts: The case study of kyber, IEEE Transactions on Computers, 71(9), pp. 2163–2176, 2021.

APPENDIX A

PYTHON IMPLEMENTATION

In this section, we present the Python implementation of toy examples designed for the following algorithms.

- Algorithm 13 - Bit-sliced binary search method proposed by [14]. See the code A.1.
- Algorithm 14 - Double and Check Algorithm. See the code A.2.
- Algorithm 15 - Integration of the Look-Up-Table (LUT). See the code A.3.3.

A.1 Python Implementation of the Algorithm 13

```
import numpy as np
import random
import math
import time

def split_into_random_shares(coefficient, num_shares, q):
    shares = [random.randint(0, q) for _ in range(num_shares-1)]
    last_share = (coefficient - sum(shares)) % q
    shares.append(last_share)
    return shares

def offset_and_compress(share, q, k):
    offset_share = (share + math.floor(q/4)) % q
    return [int(x) for x in '{0:0{1}b}'.format(offset_share, k)]

def compress_s(bits):
    x11, x10, x9, x8, x7 = bits[0], bits[1], bits[2], bits[3], bits[4]
    not_x11 = 1 - x11
    not_x8 = 1 - x8
```

```
    return x11 | (not_x11 & x10 & x9 & (x8 | (not_x8 & x7)))

def bit_sliced_binary_search(shares):
    results = []
    for bits in shares:
        results.append(compress_s(bits))
    return results

def main_algorithm(coefficient, q, num_shares):
    k = math.ceil(math.log2(q))
    shares = split_into_random_shares(coefficient, num_shares, q)
    print("The shares are:", shares)
    print("The sum of shares mod q is:", sum(shares) % q)

    offset_shares = [(share + math.floor(q/4)) % q for share in shares]
    print("The shares after offset:", offset_shares)

    compressed_shares = []
    for share in shares:
        compressed_shares.append(offset_and_compress(share, q, k))

    results = bit_sliced_binary_search(compressed_shares)
    return results

coefficient = 2419
q = 3329
num_shares = 4
start = time.time()
result = main_algorithm(coefficient, q, num_shares)
end = time.time()
print("Time result:",end - start)
print("Bit sliced binary search result:", result)
```

A.2 Python Implementation of the Double and Check Algorithm as the Toy Example

```
import numpy as np
import random
import math
import time

def split_into_random_shares(coefficient, num_shares, q):
    shares = [random.randint(0, q) for _ in range(num_shares-1)]
    last_share = (coefficient - sum(shares)) % q
    shares.append(last_share)
    return shares
```

```
11
12 def offset_share(share, q):
13     return (share + math.floor(q/4)) % q
14
15 def is_greater_than_half(share, q):
16     double_share = (2 * share) % q
17     return double_share != (2 * share)
18
19 def main_algorithm(coefficient, q, num_shares):
20     shares = split_into_random_shares(coefficient, num_shares, q)
21     print("The shares are:", shares)
22     print("The sum of shares mod q is:", sum(shares) % q)
23
24     offset_shares = [offset_share(share, q) for share in shares]
25     print("The offset shares are:", offset_shares)
26
27     results =
28     [is_greater_than_half(share, q) for share in offset_shares]
29     return results
30
31 coefficient = 2419
32 q = 3329
33 num_shares = 4
34 start = time.time()
35 result = main_algorithm(coefficient, q, num_shares)
36 end = time.time()
37 print("Time result:",end - start)
38 print("Whether shares are greater than original q/2:", result)
39
40 ************************************************************
41 The shares are: [678, 1162, 2406, 1502]
42 The sum of shares mod q is: 2419
43 The offset shares are: [1510, 1994, 3238, 2334]
44 Time result: 2.9802322387695312e-05
45 Result of whether offset shares are greater than original q/2:
46 [False, True, True, True]
47
48 ** Process exited - Return Code: 0 **
49 Press Enter to exit terminal
```

A.3 Python Implementation of the LUT Integration as the Toy Example

In this section, we will suggest an alternative way to perform a masked compression function, with an integrated Look-Up-Table (LUP).

This code piece given below (`create_lut()`) is used to create a lookup table of

size 32 with entries of type int8 (8-bit integer), which stores the results of certain bit-wise operations on the indices of the table.

A.3.1 Constructing the Look-Up-Table (LUT)

```
1  def create_lut():
2      lut = np.zeros(32, dtype=np.int8)
3      for i in range(32):
4          bits = [(i >> j) & 1 for j in range(4, -1, -1)]
5          x11, x10, x9, x8, x7 = bits
6          not_x11 = 1 if x11 == 0 else 0
7          not_x8 = 1 if x8 == 0 else 0
8          lut[i] = x11 ^ (not_x11 * x10 * x9 * (x8 ^ (not_x8 * x7)))
9      return lut
```

Notice that we choose to construct my table using the Compress_s^q function giving the Algorithm 13 introduced by [14], but this can be constructed in any other ways. As this LUT is constructed only one time before the actual compression algorithm starts, there is a trade off between memory and run-time of the algorithm.

A.3.2 The Look-Up-Table (LUT)

Below, you will see the constructed table, with a corresponding index for each 32 entity.

```
1  5-bit: 00000 : Index: 0, Value: 0
2  5-bit: 00001 : Index: 1, Value: 0
3  5-bit: 00010 : Index: 2, Value: 0
4  5-bit: 00011 : Index: 3, Value: 0
5  5-bit: 00100 : Index: 4, Value: 0
6  5-bit: 00101 : Index: 5, Value: 0
7  5-bit: 00110 : Index: 6, Value: 0
8  5-bit: 00111 : Index: 7, Value: 0
9  5-bit: 01000 : Index: 8, Value: 0
10 5-bit: 01001 : Index: 9, Value: 0
11 5-bit: 01010 : Index: 10, Value: 0
12 5-bit: 01011 : Index: 11, Value: 0
13 5-bit: 01100 : Index: 12, Value: 0
14 5-bit: 01101 : Index: 13, Value: 1
15 5-bit: 01110 : Index: 14, Value: 1
```

```
5-bit: 01111 : Index: 15, Value: 1
5-bit: 10000 : Index: 16, Value: 1
5-bit: 10001 : Index: 17, Value: 1
5-bit: 10010 : Index: 18, Value: 1
5-bit: 10011 : Index: 19, Value: 1
5-bit: 10100 : Index: 20, Value: 1
5-bit: 10101 : Index: 21, Value: 1
5-bit: 10110 : Index: 22, Value: 1
5-bit: 10111 : Index: 23, Value: 1
5-bit: 11000 : Index: 24, Value: 1
5-bit: 11001 : Index: 25, Value: 1
5-bit: 11010 : Index: 26, Value: 1
5-bit: 11011 : Index: 27, Value: 1
5-bit: 11100 : Index: 28, Value: 1
5-bit: 11101 : Index: 29, Value: 1
5-bit: 11110 : Index: 30, Value: 1
5-bit: 11111 : Index: 31, Value: 1
```

A.3.3 Python Code of the Toy Example

Below, you will see the Python implementation of an integrated LUT.

```
import numpy as np
import random
import math
import time

def split_into_random_shares(coefficient, num_shares, q):
    shares = [random.randint(0, q) for _ in range(num_shares-1)]
    last_share = (coefficient - sum(shares)) % q
    shares.append(last_share)
    return shares

def offset_and_compress(share, q):
    offset_share = (share + math.floor(q/4)) % q
    return offset_share

def create_lut():
    lut = np.zeros(32, dtype=np.int8)
    for i in range(32):
        bits = [(i >> j) & 1 for j in range(4, -1, -1)]
        x11, x10, x9, x8, x7 = bits
        not_x11 = 1 if x11 == 0 else 0
        not_x8 = 1 if x8 == 0 else 0
        lut[i] = x11 ^ (not_x11 * x10 * x9 * (x8 ^ (not_x8 * x7)))
    return lut

```

```
26 def lut_based_comparison(shares, lut):
27     results = []
28     for bits in shares:
29         index = (bits >> 7) & 0x1F  # Extract the 5-bit index
30         results.append(lut[index])
31     return results
32
33 def main_algorithm(coefficient, q, num_shares):
34     shares =
35     split_into_random_shares(coefficient, num_shares, q)
36     print("The shares are:", shares)
37     print("The sum of shares mod q is:", sum(shares) % q)
38
39     compressed_shares =
40     [offset_and_compress(share, q) for share in shares]
41     print("Shares after shifting q/4 are:", compressed_shares)
42
43     results = lut_based_comparison(compressed_shares, lut)
44     return results
45
46 coefficient = 2419
47 q = 3329
48 num_shares = 4
49 lut = create_lut() #assume that it is pre-computed.
50
51 start = time.time()
52 result = main_algorithm(coefficient, q, num_shares)
53 end = time.time()
54 print("Time result:", end - start)
55 print("Bit sliced binary search result:", result)
```