Methods for Masking CRYSTALS-Kyber Against Side-Channel Attacks

Sıla Özeren
Institute of Applied Mathematics
Middle East Technical University
Ankara, Turkey
sila.ozeren@metu.edu.tr

Oğuz Yayla
Institute of Applied Mathematics
Middle East Technical University
Ankara, Turkey
oguz@metu.edu.tr
0000-0001-8945-2780

***Abstract*—In the context of post-quantum secure algorithms like *CRYSTALS-Kyber*, the importance of protecting sensitive polynomial coefficients from side-channel attacks is increasingly recognized. Our research introduces two alternative masking methods to enhance the security of the compression function in Kyber through masking. Prior to this, the topic had been addressed by only one other research study. The "Double and Check" method integrates arithmetic sharing and symmetry adjustments, introducing a layer of obfuscation by determining coefficient values based on modular overflows. In contrast, the Look-Up-Table (LUT) integration method employs arithmetic-to-Boolean conversions, augmented by a pre-computed table for efficient value verifications. Furthermore, by leveraging the alternative prime 7681, we propose a novel masked compression function. This prime, 7681, is also notable as the smallest prime suitable for fast NTT multiplication. While both algorithms prioritize data protection and streamlined processing, they also underscore the inherent challenges of balancing computational speed with the potential vulnerabilities to side-channel attacks.**

***Index Terms*—Masking, CRYSTALS-Kyber, Post Quantum Cryptography, Side Channel Attacks**

I. Introduction

As we transition into the quantum computing era, the security of many commonly used cryptographic algorithms faces a significant challenge. Quantum computers have the capability to solve complex mathematical problems that threaten the safety of traditional cryptosystems such as RSA, DSA, and elliptic curve cryptosystems. Recognizing this, NIST initiated the Post-Quantum Cryptography (PQC) standardization process [1]. Although *Round 3* has recently concluded, many studies regarding to side-channel attacks have emerged, targeting Kyber's implementation methods in both software and hardware [2]–[4].

Drawing from the insights of foundational studies like [5] and [6], which underscore the importance of *masking* against side-channel attacks, researchers has integrated this protective measure across CRYSTALS-Kyber's components. Prior studies has already masked polynomial operations, especially the *NTT multiplication* [7], along with basic operations like addition and subtraction detailed in Alg. 5. Additionally, symmetric operations, specifically the hash function G (See Alg. 4 and PRF (See Alg. 5), have been enhanced with the Keccak masking method as outlined in [8]. For sampling tasks involving the central binomial distribution (CBD_η) in Alg. 5, the methodology from [9], which is further elaborated in [10], is employed to ensure masking.

While many components of Kyber are masked, there had been no effort to mask the compression function (see Eq. 3) used in the CPAPKE.Dec() algorithm of Kyber. This was addressed in the work published by [10], which introduced a bit-sliced binary search method on randomly generated sharings of sensitive polynomial coefficients. Building upon this study [10], we propose two alternative approaches for masking the compression function in Kyber: the *Double and Check* method, adding an additional obfuscation layer, and the classic memory/time trade-off using *Look-Up-Table* (LUT) integration.

In Section II, we provide essential information about Kyber and masking that will be referenced throughout the paper. In Section III, we discuss existing techniques related to masking compression functions and highlight how conventional methods are ill-suited for Kyber due to its prime modulo design choice. In Section IV, we present the *Double and Check* (See Alg. 1) and *Look-Up-Table* integration (See Alg. 2) method as alternatives to masking the compression function in Kyber. Additionally, we demonstrate how choosing 7681 as a prime provides a fast and masked compression function. And finally, in Section V, we present the conclusion, with areas that may rise concern or require further studies.

II. Preliminaries

In this chapter, we are going to provide the notations and definitions regarding to protection against side-channel attacks.

A. Ring of Polynomials

CRYSTALS-Kyber is a post-quantum key encapsulation mechanism (KEM) that works in the ring of polynomials over the finite field $\mathbb{Z}_q$. This ring is denoted by $R_q = \mathbb{Z}_q[x]/\langle x^n + 1\rangle$. As given in the Table I, Kyber uses a prime module $q = 3329$, causing a non-negligible performance overhead compared to modulo 2^k as studied in [11], and $n = 256$, meaning that Kyber works with polynomials that have 256 coefficients, each of them taken from the finite field $\mathbb{Z}_{3329}$.

B. Compression and Decompression in Kyber

In CCA-secure CRYSTALS-Kyber, a compression function is defined [12], which takes an element $x \in \mathbb{Z}_q$ as input and

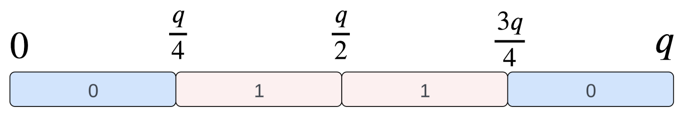

Fig. 1. Two Disjoint Interval to Compress a Polynomial Coefficient into a Single Bit [14]

returns an integer within the range $\{0,\ldots,2^d-1\}$, where $d < \lceil \log_2(q) \rceil$.

$$\textbf{Compress}_q(x,d) = \lceil (2^d/q)\cdot x \rfloor \bmod^+ 2^d \tag{1}$$

The core motivation behind this function, as the name gives a hint, is to *omit the low-order bits* of ciphertext (i.e., shortening the ciphertext) that do not significantly affect the decryption failure. Additionally, it leverages the LWE *error correction* advantages during both encryption and decryption [13].

Authors [12] also define a decompressing function such that

$$x' = \textbf{Decompress}_q(\text{Compress}_q(x,d),d), \tag{2}$$

where x' is an element in close proximity to x, more specifically, $|x' - x \bmod^{\pm} q| \leq B := \lceil \frac{q}{2^{d+1}} \rfloor$.

In the case of $d = 1$, the $\text{Compress}_q(x, d = 1)$ function 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, result of the $\text{Compress}_q(x,1)$ function can be either 0 or 1.

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 (See Eq. 3).

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

The 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 area coloured in *pink* in Fig. 1), the compression function assigns the value 1. For coefficients in the second interval $[0, \frac{q}{4}]$ and $[\frac{3q}{4}, q]$ (*blue* area in the Fig. 1), the compression function assigns the value 0.

C. Kyber Public Key Encryption

Kyber is a *module*-Learning-with-Error (LWE) scheme [15], which can be parameterized with a given security parameter k (see Table I). When the security parameter k is set to 2 or 3, we are dealing with a *module*-LWE problem (See the Lines 4-8 in Alg. 5). In other words, the notation $k > 1$ refers to the number of rings used in the module-LWE setting. Thus, when $k = 1$, it indicates that there is one ring, and it becomes a ring-LWE problem.

Security of Kyber depends on the difficulty of distinguishing between pairs, $(\mathbf{a}_i, b_i) \in \mathbb{R}_q^k \times \mathbb{R}_q$. While some of the pairs are uniformly sampled from the ring of polynomials $\mathbb{R}_q$, others are constructed such that $\mathbf{a}_i$ selected from a uniform distribution and the secret vector $\mathbf{s}$ is taken from a *central binomial distribution (CBD)* with given η (See Table I) [10], $\mathbf{s} \in \beta_\eta^k$, and some noise with small coefficients $e_i \in \beta_\eta$ such that $b_i = \mathbf{a}_i^T\mathbf{s} + e_i$.

TABLE I
Parameters of Kyber with three levels of security, k.

	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}

D. Masking as a Side-Channel Attack Countermeasure

At its core, *masking* involves randomly splitting a secret value into multiple shares. These shares are processed independently throughout the algorithm's steps, and the results are then recombined to produce the desired outcome. Working within the *masking* domain prevents information leakage from a sensitive variable a since it is never used directly. Instead, side-channel leakage arises only from calculations involving a_1 or a_2 [16]. Given that both arithmetic and Boolean shares of a are randomly defined each time, any leakage from a_1 and a_2 doesn't expose a to attackers even if the attacker runs the algorithm multiple times. As highlighted before, there are two types of masking: *Boolean* and *arithmetic*.

Consider a sensitive value $a \in \mathbb{Z}_q$. *Arithmetical masking* randomly splits this sensitive value into n arithmetic shares, which can be denoted as $\mathbf{n}_s$. The defining characteristic is that the summation of these shares results in the original value a, situated within the set of integers modulo q [10]. For $i \in \{0, 1 \cdots \mathbf{n}_s - 1\}$, $a^{(i)}$ denotes the specific share indicated by the number i.

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

When the sensitive value $\mathbb{Z}_q$ is split into $\mathbf{n}_s$ shares, we can represent a as $\mathbf{n}_s$ tuple. This tuple is given by $a^{(\cdot)_A} = (a^{(0)_A}, \cdots a^{(n_s-1)_A})$. In the *Boolean masking* case,

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

the $\mathbf{n}_s$-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 arithmetic shares $a^{(i)_B}$ with $0 \leq i < n_s$.

III. Existing Methods for Masked Compression

In this section, we discuss the reasons why the compression function introduced in Algorithm 3 in Kyber (Refer to Line 4) requires masking. We will also delve into the workings of the *Most Significant Bit* (MSB) approach. Furthermore, we'll explore how Kyber's choice of a *prime modulo* design restricts us from employing this approach to mask the compression function, as illustrated in Eq. 3. Finally, we will introduce a novel *bit-sliced binary search* method, as proposed by [10].

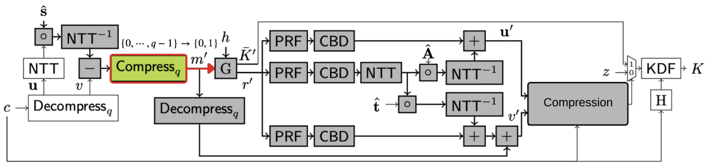

Fig. 2. 1-Bit Conversion for Polynomial Coefficient in Kyber, Updated from [10]

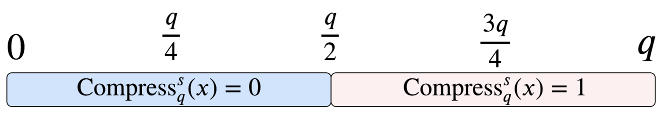

Fig. 3. Shifting the Polynomial Coefficient with Certain Offset to Mask It.

A. *Need for a Masked Compression Algorithm in Kyber*

To achieve the CCA2 security, CRYSTALS-Kyber [12] algorithm leverages the Fujisaki-Okamoto (FO) transformation [17], [18]. It first performs CPA decryption on the ciphertext (See Line 4 in Alg. 4). Then, the message is "*re*-encrypted" by CPA encryption (See Line 6 in Alg. 4), resulting in ciphertext c'. Finally, the algorithm checks whether the public ciphertext c equals c'. If $c = c'$, the algorithm turns *True*, and *False* otherwise. Depending on the *Boolean* result, the session key K is generated (See Lines 7-10 in Alg. 4). The FO-transform is done to see if an adversary performs any alteration during the process.

Any steps that directly processes the secret vector $\mathbf{s} \in \mathbb{R}_q^k$ should be masked against side-channel attacks. Thus, the steps that are coloured grey in the Fig. 2 should be masked. Up until the study done by [10], there was no research that proposed a masking for the compression function in Kyber.

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

As it is seen in the Eq. 4, the Alg. 3 directly processed the secret key $\mathbf{s} \in \mathbb{R}_q^k$. As the vector $\mathbf{u} \in \mathbb{R}_q^k$ and polynomial $v \in \mathbb{R}_q$ is publicly known, adversaries can easily do reverse-engineering to find the secret vector $\mathbf{s}$. Therefore, the compression function needs to be masked [10].

B. *The Most Significant Bit (MSB) Method*

In the most trivial approach for masking in modulo $q = 2^k$, the coefficient domain is *shifted* by a specific *offset* to form a new interval composed of two equal sub-intervals. This is done by adding $\lfloor \frac{q}{4} \rfloor$ in mod q to each coefficient share. Once each interval has been shifted by, $\lfloor \frac{q}{4} \rfloor$, a symmetry is established around $\lfloor \frac{q}{2} \rfloor$ as illustrated in Fig. 3. The new $\text{Compress}_q(x, 1)$ function can be seen in the Eq. 5.

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

$$\text{Compress}_q(x, 1) = \text{Compress}^s_q(x + \lfloor \frac{q}{4} \rfloor \mod q) \quad (5)$$

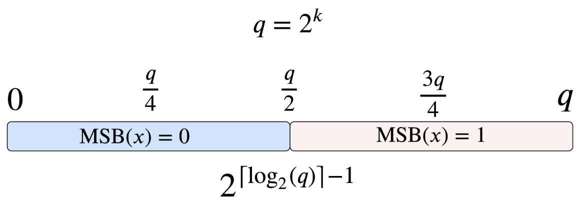

Fig. 4. Saber ($q = 2^k$), Shifted Compress^s_q Function.

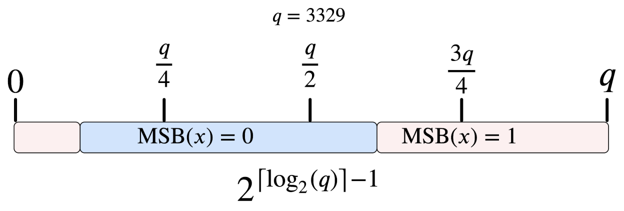

Fig. 5. Offset Between the Most Significant Bit (MSB) and $\lfloor \frac{q}{2} \rfloor$

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 trivial shifting method can be efficiently implemented. This is because, the most significant bit of a sensitive coefficient x will be a great indicator to tell whether the coefficient will fall into the first half of a coefficient domain, such as $\text{MSB}(x) = 0$ in the case where $x < \lfloor \frac{q}{2} \rfloor$. 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) over $\lfloor \frac{q}{2} \rfloor$ as it is given Fig. 5. Thus, the most significant bit (MSB) approach does not work for the Kyber. The offset arises because CRYSTAL-Kyber employs a prime number $q = 3329$. In this context, the value of the MSB (2^{11}) equates to "2048" in the coefficient domain, which can be seen in the following: $2^{\lceil log_2(q) \rceil - 1} = 2^{\lceil log_2(3329) \rceil - 1} = 2^{\lceil 11.7 \rceil - 1} = 2^{12-1} = 2^{11} = 2048$. As $\lfloor q/2 \rfloor = \lfloor 3329/2 \rfloor = \lfloor 1664.5 \rfloor = 1664$, the offset is $2048 - 1664 = 832$. 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.

C. *Bit-Sliced Binary Search Method*

An eye opening study regarding to masking the compression function in Kyber is done by [10], implementing a *bit-sliced binary search* over the 12-bits length Boolean version of an arithmetically shifted ($\lfloor \frac{q}{4} \rfloor$) share to decide whether the share is greater than equal to $\lfloor \frac{q}{2} \rfloor$. The algorithm is provided to be $t-$SNI secure with $\mathbf{n}_s = t + 1$ shares [10].

This algorithm takes each 256 coefficient of a polynomial $a \in \mathbb{Z}_q[X]$ and splits each coefficient into its $\mathbf{n}_s$ shares. Each share is shifted by $\lfloor \frac{q}{4} \rfloor$ to make the interval symmetric around

$\lfloor \frac{q}{2} \rfloor$ (See Fig. 3), making the decision of whether a share is greater than equal to $\lfloor \frac{q}{2} \rfloor$ much more easier. Then, each share is cast to its 12-bit Boolean version. $k = \lceil log_2(q) \rceil = \lceil log_2(3329) \rceil \approx \lceil 11.7067 \rceil = 12$

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. Then, the algorithm uses the following bit-wise $\text{Compress}_q^s(x)$ function to decide whether the share is greater than equal to $\lfloor \frac{q}{2} \rfloor$.

$$\text{Compress}_q^s(x) = x_{11} \oplus (\neg x_{11} \cdot x_{10} \cdot x_9 \cdot (x_8 \oplus (\neg x_8 \cdot x_7))) \tag{6}$$

If $x_{11} = 1$, then $2^{11} = 2048 > 1664$, and the coefficient should be compressed to 1. If $x_{11} = 0$, we need to consider the remaining bits. If $x_{10} = 1$ and $x_9 = 1$, then the share 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. The output of this algorithm looks like a vector depending on the number of shares, and is used by other masked components in Kyber.

IV. Contribution

In this section, we are going to present two alternative algorithms to masking the compression function in Kyber: *Double and Check* (See Alg. 1) and *Integration of Look-Up-Table (LUT)*.

A. Double and Check Algorithm

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 [10]. 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 $\lfloor q/2 \rfloor$ (See Fig. 3). Retaining the symmetric interval property of the coefficient domain 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 symmetry is achieved, the algorithm adds each shifted share to itself. This addition acts as an obfuscated method for indirectly gauging if the shifted share, when summed with itself, 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 share after summation and then taking modulo q is different from the raw summed share (before taking modulo). If these two values are not the same, an overflow has occurred.

Hence, the "*Check*" in Double and Check (See Alg. 1) is the procedure of observing whether this overflow took place.

Algorithm 1 Double and Check algorithm

```
Require: 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 $a_i^{(\cdot)_A} = a_i^{(0)_A} + a_i^{(1)_A} + \cdots + a_i^{(n_s-1)_A} \mod q$.
Ensure: 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:         $a_i^{(k)_A} \leftarrow a_i^{(k)_A} + \lfloor \frac{q}{4} \rfloor \bmod q$
 4:         raw_double_share $\leftarrow a_i^{(k)_A} + a_i^{(k)_A}$
 5:         double_share $\leftarrow$ raw_double_share mod $q$
 6:         if double_share = raw_double_share then
 7:             $m_i^{(k)_B} \leftarrow 0$
 8:         else
 9:             $m_i^{(k)_B} \leftarrow 1$
10:         end if
11:     end for
12: end for
13: return $m^{(\cdot)_B}$
```

When overflow is detected, the algorithm outputs *False* with value 0, indicating the original share was below or equal to the midpoint $\lfloor \frac{q}{2} \rfloor$. If overflow occurred, the output is *True* with value 1, indicating the original share was above the midpoint $\lfloor \frac{q}{2} \rfloor$. The result will be an array of $\mathbf{n}_s$ bits for each coefficient.

B. Integration of Look-Up-Table (LUT)

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. The LUT-based method uses one *Arithmetic-to-Boolean* conversion per each share (See Alg. 2).

The algorithm takes each sensitive coefficient of a polynomial $x \in \mathbb{Z}_{3329}[X]$ as input and splits it into $\mathbf{n}_s$ shares. Each share is shifted by $\lfloor \frac{q}{4} \rfloor = 832$. This is again done to my the interval symmetric around $\lfloor \frac{q}{2} \rfloor = 1664$. Subsequently, every share is converted into its 12-bit Boolean representation (*Arithmetic-to-Boolean*). This representation is then used to determine if the share is greater than or equal to 1664, a decision facilitated by a Look-Up-Table (LUT).

Therefore, since only the 5 most significant bits of a number are needed to check whether it is greater than or equal to $\lfloor \frac{q}{2} \rfloor = 1664_{10} = (\mathbf{01101}0000000)_2$, the algorithm requires $2^5 = 32$ entities (See Table II).

To illustrate the process, consider a coefficient share $2035 \in \mathbb{Z}_{3329}$, which is already shifted by $\lfloor \frac{q}{4} \rfloor$ in mod $q = 3329$. First, we convert 2035 into its binary representation, resulting in $(01111111011)_2$. From this binary form, we extract its 5 most significant bits, which are 01111 (equivalent to 15 in decimal notation). Then, by checking the value in the Look-Up-Table at index 15, we find it's 1. Consequently, the Boolean representation for the share 2035 is determined to be 1.

TABLE II
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

Algorithm 2 LUT-based Compression Algorithm

```
Require: An arithmetic sharing $a^{(\cdot)_A}$ of a polynomial $a \in \mathbb{Z}_q[X]$, $a_i^{(\cdot)_A} = a_i^{(0)_A} + a_i^{(1)_A} + \cdots + a_i^{(n_s-1)_A} \mod q$.
Ensure: A Boolean sharing $m^{(\cdot)_B}$ indicating if each offset share is greater than $q/2$.
 1: Pre-compute LUT for indices 0-31, each being a binary representation of an integer $i$ with bits $x_{11}, x_{10}, x_9, x_8, x_7$.
 2: for $i = 0$ to 255 do    ▷ Loop through all 256 coefficients
 3:     for $k = 0$ to $n_s - 1$ do ▷ Loop through all shares for each coefficient
 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:         $m_i^{(k)_B} \leftarrow \text{LUT[index]}$
 8:     end for
 9: end for
10: return $m^{(\cdot)_B}$
```

C. Alternative Prime Number for Fast Compression Function

In this session, we examined the case where the prime number used in Kyber is set to 7681, instead of 3329. If we consider the scenario with $q = 3329$, we observe that it utilizes the 5 most significant bits of $\lfloor \frac{3329}{2} \rfloor$ with 4 AND, 2 NEG, and 1 XOR operations. However, when $q = 7681$, the new masked compression function (See Eq. (7)) becomes 1 NEG and 1 XOR operation less expensive than the function in Eq. (6), which can be seen from the Table III.

$$\lfloor \frac{7681}{2} \rfloor = 3840$$
$$3840 = (0\mathbf{1111}00000000)_2$$
$$\text{Compress}_q^s(x) := x_{13} \oplus (\neg x_{13} \cdot x_{12} \cdot x_{11} \cdot x_{10} \cdot x_9) \quad (7)$$

In fact, the prime 7681 is the smallest prime that satisfies $q \equiv 1 \pmod{2n}$ such that $n = 256$. Polynomial multiplication is one of the most computationally intensive parts of lattice-based cryptographic schemes. The NTT algorithm relies on the existence of a $2n$-th primitive root of unity in the finite field of integers modulo q. Hence, the prime number 7681 is not only suitable for fast NTT multiplication but also provides a fast masked compression function.

TABLE III
ANALYSIS FOR THE PRIME NUMBER 7681

Attribute	**Value**
Number	7681
k	14
$\lfloor \frac{q}{2} \rfloor$	3840.5 $\simeq$ 3840
Binary representation of 3840	$(0111100000000)_2$
$\text{Compress}_q^s(x)$	$x_{13} \oplus (\neg x_{13} \cdot x_{12} \cdot x_{11} \cdot x_{10} \cdot x_9)$
Number of XOR operations	1
Number of AND operations	4
Number of NEG operations	1

V. CONCLUSION

In conclusion, our study delves into alternative methods for ensuring the security of polynomial coefficients during compression in CRYSTALS-Kyber, focusing on safeguarding against side-channel attacks.

The presented "Double and Check" algorithm, along with the LUT-based method, provide effective means of coefficient masking, each with its distinct advantages and disadvantages. Additionally, we have proposed a fast masked compression function for the CPAPKE.Dec() algorithm using the prime number 7681, which also facilitates fast NTT multiplication.

While the *Double and Check* method preserves integrity and promotes computational efficiency, it does necessitate *modular addition*, which is more computationally intensive than a bitwise operation proposed in [10]. But more efficient algorithms exist (like *Barrett reduction* [19]) that can reduce this complexity. However, these reductions may present additional risk for side-channel leakage, as shown in the work [20].

In terms of Look-Up-Table integration, which utilizes modular arithmetic and pre-computed tables, these methods not only ensure enhanced data protection but also provide computational efficiency. However, they do require a trade-off between memory and computational efficiency. Even though the table is quite small, in scenarios involving hardware implementations with limited memory, this may pose a concern.

While both algorithms significantly enhance the security paradigm, it's crucial to weigh the potential trade-offs in specific use-case scenarios. As we navigate an ever-evolving landscape of cybersecurity threats, this research highlights the pressing need for continuous innovation in data protection techniques.

ACKNOWLEDGMENT

The first author gratefully acknowledges the support of TÜBİTAK's "2210-A National MSc/MA Scholarship Program." We extend our heartfelt appreciation to TÜBİTAK for their invaluable contribution to academic research.

REFERENCES

[1] N. I. of Standards and Technology, "Pqc standardization process: Announcing four candidates to be standardized, plus fourth round candidates," 2022, information Technology Laboratory, COMPUTER SECURITY RESOURCE CENTER. [Online]. Available: https://csrc.nist.gov/News/2022/pqc-candidates-to-be-standardized-and-round-4

[2] 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*, 2023, pp. 10–20.

[3] 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*, vol. 71, no. 9, pp. 2163–2176, 2021.

[4] 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)*. IEEE, 2023, pp. 1–5.

[5] 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*. Springer, 1999, pp. 398–412.

[6] 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*. Springer, 2013, pp. 142–159.

[7] 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.

[8] 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*, 2016, pp. 116–129.

[9] 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*. Springer, 2019, pp. 534–564.

[10] 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.

[11] 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*. Springer, 2019, pp. 344–362.

[12] 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)*. IEEE, 2018, pp. 353–367.

[13] 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*, vol. 2, no. 4, pp. 1–43, 2019.

[14] 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]. [Online]. Available: https://www.youtube.com/watch?v=R6lNR3ihVuU&t=230s

[15] Z. Brakerski, C. Gentry, and V. Vaikuntanathan, "(leveled) fully homomorphic encryption without bootstrapping," *ACM Transactions on Computation Theory (TOCT)*, vol. 6, no. 3, pp. 1–36, 2014.

[16] 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)*, vol. 17, no. 2, pp. 1–26, 2021.

[17] E. Fujisaki and T. Okamoto, "Secure integration of asymmetric and symmetric encryption schemes," in *Annual international cryptology conference*. Springer, 1999, pp. 537–554.

[18] D. Hofheinz, K. Hövelmanns, and E. Kiltz, "A modular analysis of the fujisaki-okamoto transformation," in *Theory of Cryptography Conference*. Springer, 2017, pp. 341–371.

[19] W. Hasenplaugh, G. Gaubatz, and V. Gopal, "Fast modular reduction," in *18th IEEE Symposium on Computer Arithmetic (ARITH'07)*. IEEE, 2007, pp. 225–229.

[20] B.-Y. Sim, A. Park, and D.-G. Han, "Chosen-ciphertext clustering attack on crystals-kyber using the side-channel leakage of barrett reduction," *IEEE Internet of Things Journal*, vol. 9, no. 21, pp. 21 382–21 397, 2022.

VI. Appendix

Algorithm 3 Kyber.CPAPKE.Dec(c, sk): decryption [13]

```
Require: Secret key $sk \in B^{12 \cdot k \cdot n/8}$
Require: Ciphertext $c \in B^{d_u \cdot k \cdot n/8 + d_v \cdot n/8}$
Ensure: 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 $m$
```

Algorithm 4 Kyber.CCAKEM.Decaps(c, sk): decryption [13]

```
Require: Ciphertext $c \in B^{d_u \cdot k \cdot n/8 + d_v \cdot n/8}$
Require: Secret key $sk \in B^{24 \cdot k \cdot n/8 + 96}$
Ensure: 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:     $K \leftarrow \text{KDF}(\overline{K}' || H(c))$
 9: else
10:     $K \leftarrow \text{KDF}(z || H(c))$
11: end if
12: return $K$
```

Algorithm 5 Kyber.CPAPKE.Enc(pk, m, r): encryption [13]

```
Require: Public key $pk \in \mathcal{B}^{12 \cdot k \cdot n/8 + 32}$
Require: Message $m \in \mathcal{B}^{32}$
Require: Random coins $r \in \mathcal{B}^{32}$
Ensure: 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 $B_{\eta_1}$
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 $B_{\eta_2}$
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 $c$          ▷ $c \leftarrow (\text{Compress}_q(\mathbf{u}, d_u), \text{Compress}_q(v, d_v))$
```