Quasi-linear Masking to Protect Kyber against both SCA and FIA[⋆]

Pierre-Augustin Berthet[1,2][0009−0005−5065−2730] (✉), Cédric Tavernier[2][0009−0007−5224−492X], Jean-Luc Danger[1][0000−0001−5063−7964], and Laurent Sauvage[1][0000−0002−6940−6856]

[1] Télécom Paris, 19 Place Marguerite Perey, F-91123 Palaiseau Cedex, France
{(✉)berthet,jean-luc.danger,laurent.sauvage}@telecom-paris.fr
[2] Hensoldt SAS France, 115 Avenue de Dreux, 78370 Plaisir, France
{pierre-augustin.berthet,cedric.tavernier}@hensoldt.net

Abstract. The recent technological advances in Post-Quantum Cryptography (PQC) rise the questions of robust implementations of new asymmetric cryptographic primitives in today's technology. This is the case for the lattice-based CRYSTALS-Kyber algorithm which has been selected as the first NIST standard for Public Key Encryption (PKE) and Key Encapsulation Mechanisms (KEM). We have notably to make sure the Kyber implementation is resilient against physical attacks like Side-Channel Analysis (SCA) and Fault Injection Attacks (FIA). To reach this goal, we propose to use the masking countermeasure, more precisely the generic Direct Sum Masking method (DSM). By taking inspiration of a previous paper on AES, we extend the method to finite fields of characteristic prime other than 2 and even-length codes. In particular, we investigated its application to Keccak, which is the hash-based function used in Kyber. We also provided the first masked implementation of Kyber providing both SCA and FIA resilience while not requiring any conversion between different masking methods.

Keywords: Post-Quantum Cryptography · CRYSTALS-Kyber · Side Channel Analysis · Fault Injection Attack · Masking · Direct Sum Masking

1 Introduction

Since the dawn of cryptology, cryptanalysis has focused on the theoretical background used to perform cryptography. However, since the late 1990s and the publication of Kocher on side channel analysis (SCA) [22], physical attacks try to take advantage of leakages or faults within the implementation rather than breaking the algorithm in itself. For this reason, the software and hardware designers of cryptographic primitives have to take into account this threat. The recent Post-Quantum Cryptographic algorithms are particularly targeted as their implementation still requires secure architectures and analysis to make them robust against physical attacks.
Quantum computing is an active research field which progresses monthly and the likelihood of an efficient quantum computer in the coming 30 years is almost certain [23]. Such a computer would be able to break current asymmetric cryptography primitives by taking advantage of the Shor quantum algorithm [29]. In order to assure a continuity in asymmetric cryptography, the NIST has launched a standardization process of PQC in 2016 [12] resulting in an international competition to create the future digital signature, PKE and KEM protocols which must be secure

[⋆] Supported by Agence de l'Innovation de Défense, Ministère des Armées against quantum and classical computer. The end of the third and final round was announced the 5th of July 2022 [1] and 3 signatures and one PKE/KEM were selected while 4 other KEMs are heading for a final round to serve as alternatives in case of a cryptanalysis breakthrough [3]. The selection process focused first on quantum resilience, cost and performance, and then on the algorithm and its implementation. Most of the candidates claimed to be secure against time-based SCA as they provide constant time implementation and no conditional branching depending on sensitive data. But they do not make them secure against power-based SCA, like like Correlation Power Analysis (CPA), and Fault Injection Attacks (FIA). Even more, some of the candidates contains functions that can not be easily secured using generic defenses and will require specific mechanisms to ensure their side channel resilience.

1.1 Background on Masking

One of the most efficient and proven countermeasure against power-based SCA is masking [11]. The core idea is to avoid manipulating the sensitive data but instead "shares" of it that will be reassembled after the computations are done. The shares being a combination of the sensitive data and a number of random variables called masks. Thus, an attacker will only observe leakages from the shares and might not be able to recover the secret data. The order of masking is determined by the number of independent shares used. A high-order of masking means a better security against differential attacks but it generally comes at the cost of performances and space. Classical masking involves either arithmetical masking, where the random shares are subtracted or added to the secret, and boolean masking where the random shares are XORed with the secret. Conversions from one type of masking to the other do exist but have to be performed carefully. Here we will use a generic code-based masking scheme called Direct Sum Masking (DSM), introduced by Bringer et al. [8].
In this paper we focus on CRYSTALS-Kyber [6], a post-quantum PKE/KEM. They have been already several publications on how to mask it on several platforms. Most noticeably, the work from Heinz et al. [19] proposed the first open-source implementation of a masked Kyber on microprocessor while relying on the work of Oder et al. [25] on previous lattice-based primitives. Bos et al. [7] proposed a masked software implementation of Kyber while Bronchain and Cassiers [9] proposed new gadgets for Arithmetic to Boolean (A2B) and B2A conversions and tested them in a open-source masked implementation of CRYSTALS-Kyber for microprocessors. When it comes to other platforms, Fritzmann et al. [14] worked on masking HW/SW codesign. Beckwith et al. [4] worked on a shared FPGA implementation of CRYSTALS-Kyber and CRYSTALS-Dilithium while masking the CRYSTALS-Kyber. Finally, Pöppelmann and Heinz [20] proposed a combined fault and DPA protection for lattice-based cryptography, however they only secured the arithmetic parts of the algorithm.

Remark 1. It is important to note that masking at the first order alone is not a sufficient defense. The PhD thesis work of Kalle Ngo [24] and master's thesis of Linus Backlund [3] (both from KTH Stockholm) proved that novel methods relying on deep-learning were able to thwart attempts of protecting Kyber with first order masking and/or shuffling. Hence, it is important to either mix defense mechanism (shuffling, blinding, hiding...) or use higher order masking. Also note that these attack methods have not been tested yet against code-based masking.

[3] One of the KEMs of the 4th round fell victim of such a breakthrough in August 2022, stressing the need for alternative standards and hybridization

1.2 Our contributions

The main contributions we are making with this paper are as follow:

- We extend the code-based masking method from [10] to work over prime modulus finite fields and with even-length codes.
- We propose to adapt this code-based masking method to the post-quantum PKE/KEM Kyber.
- We use the same method to mask a Keccak implementation which will be used inside Kyber.
- To the best of our knowledge, we are the first to propose a masking method over this post-quantum KEM that does not require any change or conversion in terms of masking method.
- This method allows not only to mask at high order but also offer an error correcting capability, thus reducing the impact of Fault Injection Attack while our masking is active, albeit at the cost of regular calls to the error detection mechanism.
- Finally we discuss performances of our masking method and possible axis to improve them.

The paper is structured as follows: in Section 2, we introduce notations and CRYSTALS-Kyber. In Section 3, we present our Direct Sum Masking. In Section 4, we explain how we adapted our masking method to CRYSTALS-Kyber. Finally, in Section 5, we discuss performances and possible improvements for our design. Section 6 concludes our paper.

2 Preliminaries

2.1 Notations

Let $n \in \mathbb{N}^*$ the length of a code and k its dimension. We denote *odm* the masking order. We consider the finite field $\mathbb{F}_q$ with q a prime integer. Let ν a primitive element of $\mathbb{F}_q$. We assume that $n \neq 0 \mod q$ divides $q-1$, then we have

$$\omega = \nu^{\frac{q-1}{n}} \Rightarrow \omega^n = 1.$$

We must distinguish the case n odd and n even, then we set $d = \lfloor n/2 \rfloor$. For any vector $(u_0, ..., u_{n-1}) \in \mathbb{F}_q^n$, we can associate the polynomial $U(X) = u_0 + u_1 X + \ldots + u_{n-1} X^{n-1}$ and the discrete Fourier transform is defined by

$$\mathtt{DFT}_\omega(u_0, ..., u_{n-1}) = \left(\sum_{i=0}^{n-1} u_i \omega^{ij} \right)_{j \in [0 \ldots n-1]} = \left(U(\omega^j) \right)_{j \in [0 \cdots n-1]} .$$

Then the $\mathtt{DFT}_\omega$ inverse is defined by:

$$\mathtt{IDFT}_\omega(U(1), \ldots, U(\omega^{n-1})) = n^{-1} \left(\sum_{i=0}^{n-1} U(\omega^i) \omega^{-ij} \right)_{j \in [0 \cdots n-1]} = (u_0, ..., u_{n-1}).$$

Remark 2. We have clearly made the hypothesis "n divides $q-1$" to find the condition of application of the Fast Fourier Transform but the procedure that we are going to develop obviously works by considering respectively $\mathtt{DFT}_\omega$ and $\mathtt{IDFT}_\omega$ as a Vandermonde multiplication and its inverse. The impact is just in term of complexity which cost n^2 multiplications over $\mathbb{F}_q$ against $\mathcal{O}(n \log n)$ for a $\mathtt{DFT}_\omega$ in the most favourable cases.

For the Kyber algorithm, the considered $\texttt{DFT}_\omega$ is coming from methods described in [10,31]. It consists in building a tree (see section 4) of polynomials and to compute input vector interpreted as a polynomial modulo these polynomials. In particular cases, for example over finite fields of even characteristic and $n+1$ a power of two, the tree is composed of linearized polynomials (up to constant) which are sparse by nature [31]. For example in the figure below, the tree is defined over the finite field $\mathbb{F}_{2^4}$ with $n = 3$ and ω satisfying $\omega^3 = 1$:

$$\begin{tikzcd}
 & & & q_{2,0} = X^4 + X \arrow[dll] \arrow[drr] & & & \\
 & q_{1,0} = X^2 + X \arrow[dl] \arrow[dr] & & & & q_{1,1} = X^2 + X + 1 \arrow[dl] \arrow[dr] & \\
q_{0,0} = X & & q_{0,1} = X + 1 & & q_{0,2} = X + \omega & & q_{0,3} = X + \omega + 1
\end{tikzcd}$$

Then, to calculate $\texttt{DFT}_\omega(C)$ with $C = (c_0, c_1, c_2, c_3)$ and $C(X) = c_0 + c_1X + c_2X^2 + c_3X^3$, we first compute $C_{1,0} = C(X) \mod q_{1,0}$ and $C_{1,1} = C(X) \mod q_{1,1}$, finally we get the result by performing $C_{1,0} \mod q_{0,0}$, $C_{1,0} \mod q_{0,1}$, $C_{1,1} \mod q_{0,2}$ and $C_{1,1} \mod q_{0,3}$. We show in the section 4 that we have the same principles with the KYBER parameters.

We have seen that the $\texttt{DFT}_\omega$ operation is equivalent to a Vandermonde matrix multiplication $V(\omega)$ with $V(\omega) = (\omega^{ij})_{i,j \in [\![0,2d-1]\!]}$ and

$$\texttt{DFT}_\omega(u_0, ..., u_{n-1}) = (u_0, ..., u_{n-1}) \times V(\omega).$$

For length n vectors of the form $(u_0, ..., u_{d-1}, 0, \dots, 0)$, the $\texttt{DFT}_\omega$ operation corresponds to an encoding procedure by the Reed-Solomon code denoted: $\mathrm{RS}[n, d, n-d+1]$. A generator matrix of this code is given by the shortened matrix $(\omega^{ij})_{i \in [\![0,d-1]\!], j \in [\![0,2d-1]\!]}$. We recall some results that can be found in [26]: This error correcting code is classic, it is a MDS (maximum distance separable) code, which means that its minimal distance is optimal and equals $n - d + 1$ where n is code length and d is its dimension. Among the good properties of these codes, we have, if R is a generator matrix of MDS code $\mathcal{R}$ of length n and dimension d that:

– If $\mathcal{R}$ is MDS, then $\mathcal{R}^\perp$ is MDS where $\mathcal{R}^\perp$ is the code defined by kernel(R);
– If R is MDS, then all set of d columns are free.

We recall that any $[n, d, n-d+1]$-linear code can detect until $n - d$ errors.

2.2 CRYSTALS-Kyber

CRYSTALS-Kyber [6] is a Module-Lattice-Based KEM which has been selected[4] at the end of the 3rd round of the NIST Standards Post-Quantum Competition in July 2022 [1]. It relies on several instances of the Module-LWE/LWR problems for its key generation, encapsulation and decapsulation procedures.
At its core, Kyber is a CPA-Secure PKE. To ensure CCA-level of security and a KEM status,

[4] We present the candidate version 3.02 [2] here. The final standard specifications might differ.

a modified version of the Fujisaki-Okamoto Transform [15] is used.
CRYSTALS-Kyber has three levels of security, with different parameter sets (see [2] Table 1 page 11 or Table 2 in Appendix B). All sets use the same modulo, namely $q = 3329$. We also denote $\mathbb{Z}_q[X]/(X^{256}+1)$ by R_q and $S_\eta := \{P \in R_q, \|P\|_\infty \leq \eta\}$ a subset of R_q.
Amongst other notations defined by Kyber, we have $\lceil \rfloor$, the nearest integer with ties rounded up used in the compression functions, defined as follow:

$$Compress_q(\alpha, d_i) = \left\lceil \frac{2^{d_i}}{q} \cdot \alpha \right\rfloor \; mod \; 2^{d_i}, \alpha \in \mathbb{Z}_q \tag{1}$$

$$Decompress_q(\beta, d_i) = \left\lceil \frac{q}{2^{d_i}} \cdot \beta \right\rfloor, \beta \in \mathbb{F}_{2^{d_i}} \tag{2}$$

When applied to a vector of polynomials, those two functions will then be applied to each coefficient of each polynomial separately.

Remark 3. It is interesting to note that, for $d_i = 1$, the $Decompress_q$ function can be seen as a simple multiplication by a scalar, as the value β in the equation 2 can be extracted from the rounding as it can only be 0 or 1. Thus, we have $\lceil \frac{q}{2} \cdot \beta \rfloor = 1665 \cdot \beta$. This does not apply to $Compress_q$ (Equation 1) however.

Remark 4. It is also important to note that the compression functions are lossy:

$$\text{If } m' = Decompress_q(Compress_q(m, d_i), d_i), \text{ then } |m - m'| \leq \lceil q/2^{d_i+1} \rfloor \tag{3}$$

In Kyber, the distribution used for random sampling of sensitive values is the Center Binomial Distribution:

$$CBD_\eta(\beta) = \sum_{i=0}^{255} (\sum_{j=0}^{\eta} \beta_{2i\eta+j} - \sum_{j=0}^{\eta} \beta_{2i\eta+\eta+j}) X^i \text{ with } \beta \in \{0,1\}^{512\eta} \tag{4}$$

This function is fed with a pseudo-random input β, generated by

$$PRF(seed, N) = SHAKE256(seed \| N)$$

The counter here allows seed reuse for the multiple values sampled during the PKE algorithms of CRYSTALS-Kyber. We will use the $\hookleftarrow$ notation for sensitive value sampling. Keep in mind this is a call to Equation 4 where the input is $PRF(seed, N)$. The N counter is incremented after each call to CBD.
Non sensitive values are sampled a bit differently but this is out of the scope of this paper and we will simply denote this sampling by $\leftharpoonup$.

We will only present the KEM Decapsulation of CRYSTALS-Kyber here. For more details, we invite you to consult Appendix B where are described the PKE and KEM algorithms as well as figures showing the sensitiveness of the different operations within CRYSTALS-Kyber. You can also consult the reference paper of CRYSTALS-Kyber [6] for the algorithms and [27] (slide 76), [28] (slide 32-35) for the sensitiveness.

Algorithm 1 KEM-Kyber Decapsulation

```
1: Input: Ciphertext $c = (c_u, c_v)$
2: Input: Secret Key $sk = (\vec{s}, pk, H(pk), z)$
3: Output: Shared key $K$
4: $\vec{u}, v = Decompress_q(c_u, d_u), Decompress_q(c_v, d_v)$
5: $m' := PKE.Decaps(\vec{s}, \vec{u}, v)$
6: $(K', seed') := G(m' \| H(pk))$
7: $\vec{u'}, v' := PKE.Encaps(pk, m', seed')$
8: $c' = (Compress_q(\vec{u'}, d_u), Compress_q(v', d_v))$
9: if $c == c'$ then
10:     return $KDF(K' \| H(c))$
11: else
12:     return $KDF(z \| H(c))$
13: end if
```

Remark 5. H, G and KDF are all different Keccak instances.

Remark 6. Keep in mind that PKE Encaps will always result in the same outputs for a given set of inputs, as the seed for the sampling is one of the inputs. Thus, tampering with the ciphertexts results in tampering with the seed and a completely different result out of the re-encapsulation.

If you are interested in knowing more about CRYSTALS-Kyber, we invite you to read the specification paper from the CRYSTALS team [6].

3 Code Based Masking, a DSM Example

The DSM encoding [8] consists in mapping the information x in a masked information (x, r) where r is a random mask such that:

$$x \mapsto (x, r) \mapsto x\mathbf{G} + r\mathbf{H}. \tag{5}$$

where $\mathbf{G}$ and $\mathbf{H}$ are two generator matrices of the two complementary codes $\mathcal{C}$ and $\mathcal{D}$ with $\mathcal{C} \cap \mathcal{D} = \{0\}$.
We propose to describe a masking method based on Reed Solomon encoding. This method is described in [10] for the characteristic 2 and odd length. We show in this section that it works for the characteristic prime q. We want to mask an information of size t and we assume that $\omega \in \mathbb{F}_q$ is a n-square root of unity and we consider a free family $u_0, u_1, u_2, ..., u_{d-1}$ of $\mathbb{F}_q^{d-1}$ with $u_i \neq \omega^j$ for any $0 < i \leq t-1$ and $0 < j \leq n-1$. We want now to mask the vector $\vec{x} = (x_0, \dots, x_{t-1}) \in \mathbb{F}_q^t$ with $t < d$ and $d = \lfloor n/2 \rfloor$.

3.1 Encoding Procedure

First we pick randomly $\vec{r} = (r_t, r_{t+1}, \dots, r_{d-1})$ in $\mathbb{F}_q^{d-t}$. It is well known that there exist a vector $\vec{a} = (a_0, a_1, \dots, a_{d-1})$ and the associate polynomial $P_{\vec{x}}(X) = a_0 + a_1 X + \cdots + a_{d-1} X^{d-1}$ of degree at most $d-1$ that satisfies $P_{\vec{x}}(u_i) = x_i$ for $i \in \{0, \dots, t-1\}$ and $P_{\vec{r}}(u_i) = r_i$ for $i \in \{t, \dots, d-1\}$.

Let us denote the matrix $A \in \mathbb{F}_q^{(d)\times(d)}$, where $A_{i,j} = u_i^j$ for any i, j in $\{0, \dots, d-1\}$. We have:

$$\vec{a} = (\vec{x} \mid \vec{r}) \times (A^{-1})^\top$$

The second of our masking procedure consists in evaluating the $P_{\vec{r}}$ over the set $1, \omega, \omega^2, \dots, \omega^{n-1}$. By construction, the second step of encoding consists in computing $\mathtt{DFT}_\omega(a_0, \dots, a_{d-1}, 0, \dots, 0)$. Thus finally:

$$\mathtt{Mask}(\vec{x}) = \mathtt{DFT}_\omega(a_0, \dots, a_{d-1}, 0, \dots, 0).$$

Algorithm 2 `SeveralByteMasking` Complexity : d^2

```
1: Input: a sensitive vector $\vec{x} \in \mathbb{F}_q^t$
2: Output: $\mathtt{Mask}(\vec{x}) \in \mathbb{F}_q^n$
3: $\vec{r} \xleftarrow{\$} \mathbb{F}_q^{d-t}$
4: $\vec{a} \leftarrow (\vec{x} \mid \vec{r}) \times (A^{-1})^\top$        ▷ $A^{-1}$ is a precomputed value
5: return $\mathtt{DFT}_\omega(\vec{a} \mid \vec{0})$
```

We have presented a $\mathcal{O}(d^2)$ complexity encoding procedure, but we can do better with the following method: We can construct $P(X) = T_t(X) + R_t(X)$ by first picking randomly the polynomial $T_t(X) = a_t X^t + \dots + a_{d-1}X^{d-1}$. Then we evaluate T_t over $1, u, \dots, u_{t-1}$ which cost $t(d-1-t)$ multiplications over $\mathbb{F}_q^{d-1}$. We want now constructing $R_t(X) = a_0 + a_1 X + \dots, a_{t-1}X^{t-1}$ which leads to solve the linear system

$$\underbrace{\begin{bmatrix} 1 & u_0 & \dots & u_0^{t-1} \\ & & \vdots & \\ 1 & u_i & \dots & u_i^{t-1} \\ & & \vdots & \\ 1 & u_{t-1} & \dots & u_{(t-1)}^{t-1} \end{bmatrix}}_{A} \times \underbrace{\begin{bmatrix} a_0 \\ \vdots \\ a_i \\ \vdots \\ a_{t-1} \end{bmatrix}}_{\vec{a}\,'} = \underbrace{\begin{bmatrix} x_0 + T_t(u_0) \\ \vdots \\ x_i + T_t(u_i) \\ \vdots \\ x_{t-1} + T_t(u_{t-1}) \end{bmatrix}}_{\vec{y}\,'}.$$

The matrix inversion of A is a precomputation, thus, the calculation of:

$$\vec{a}\,' = A^{-1}\vec{y}\,'$$

costs $(t+1)^2$ multiplications over $\mathbb{F}_q$. Hence, the total cost of this encoding (including the $T_t(u_i)$ calculation) does not exceed $t(d-1-t)+t^2 = t(d-1)$ multiplications over $\mathbb{F}_q$. Again, the second step of encoding consists in computing $\mathtt{DFT}_\omega(a_0, \dots, a_{d-1}, 0, \dots, 0)$ which can be achieved with not more than $(2d-1)\log(2d-1)$ multiplications over $\mathbb{F}_q$.

Remark 7. All the aforementioned operations are obviously reversible and we denote by `Unmask` the reverse operation. A tedious calculation gives a complexity in $t(d-1) + (2d-1)\log(2d-1)$ multiplications over $\mathbb{F}_q$.

3.2 Error Correcting Code Interpretation

We note that by construction, there exist an invertible matrix R that satisfies:

$$\begin{pmatrix} a_0 \\ \vdots \\ a_{t-1} \\ a_t \\ \vdots \\ a_{d-1} \end{pmatrix} = R \times \begin{pmatrix} x_0 \\ \vdots \\ x_{t-1} \\ P(u_t) \\ \vdots \\ P(u_{d-1}) \end{pmatrix}$$

We note that this DFT computation corresponds to the encoding in the Reed-Solomon code defined by the evaluation of $1, X, \ldots, X^{d-1}$ over $1, \omega, \omega^2, ..., \omega^{n-1}$, and represented by a Vandermonde matrix $V(\omega)$. Hence, we get that

$$\texttt{Mask}(\vec{x}) = (\vec{x}, \vec{r}) R^\top V(\omega) \quad (= \vec{x}G + \vec{r}H \text{ in the DSM model}).$$

We deduce that our masking algorithm corresponds to encoding procedure with a generalized Reed-Solomon code of minimal distance $n - d + 1$, dimension d and length n.

3.3 Masking Addition, Subtraction and Scaling

Let us denote: $\vec{z} = \texttt{Mask}(x)$ and $\vec{z}\,' = \texttt{Mask}(x')$. The following properties are obviously satisfied:

- $\texttt{Mask}(x + x') = \vec{z} + \vec{z}\,'$,
- $\texttt{Mask}(x - x') = \vec{z} - \vec{z}\,'$,
- $\texttt{Mask}(\lambda x) = \lambda \cdot \vec{z}$ for any $\lambda \in \mathbb{F}_q$.

3.4 Masking the Multiplication

Let's denote: $\vec{z} = \texttt{Mask}(\vec{x})$ and $\vec{z}\,' = \texttt{Mask}(\vec{x}')$. Obviously,

$$\vec{z} \odot \vec{z}\,' = \texttt{DFT}_\omega(a_0, \ldots, a_{d-1}, 0, \ldots, 0) \odot \texttt{DFT}_\omega(a'_0, \ldots, a'_{d-1}, 0, \ldots, 0).$$

The polynomial obtained by performing $\texttt{DFT}_\omega^{-1}(\texttt{DFT}_\omega(P_{\vec{x}}) \times \texttt{DFT}_\omega(P_{\vec{x}'})) = P_{\vec{x}}(X) \times P_{\vec{x}'}(X) = C(X)$ is a $2d-2$ degree polynomial, which satisfies $C(u_i) = P_{\vec{x}}(u_i) \times P_{\vec{x}'}(u_i) = x_i x'_i$ for i in $\{0, \ldots, t-1\}$.
Now we have to propose a method that associates a degree $d-1$ polynomial $D(X)$ to $C(X)$. This polynomial must satisfies the same properties: $D(u_i) = C(u_i)$ for all $0 \leq i \leq t-1$.
The authors of [17] proposed the following construction for $t = 1$:

$$\begin{aligned} D(X) &= c_0 + c_1 X + \ldots + c_{d-1} X^{d-1} + u_0^{d-1}(c_d X + \ldots + c_{2d-2} X^{d-1}) \\ &= c_0 + (c_1 + u_0^{d-1} c_d) X + \cdots + (c_{d-1} + u_0^{d-1} c_{2d-2}) X^{d-1} \ . \end{aligned}$$

Obviously, in this case $D(u_0) = C(u_0) = x_0 x'_0$. This construction can be generalized and let:

$$U_j(X) = u_j^{d-1} \frac{(X - u_0) \cdots (X - u_{j-1})(X - u_{j+1}) \cdots (X - u_t)}{(u_j - u_0) \cdots (u_j - u_{j-1})(u_j - u_{j+1}) \cdots (u_j - u_t)}.$$

Hence, by construction, $U_j(u_j) = u_j^{d-1}$ and $U_j(u_i) = 0 \; \forall \, i \in \{0, \ldots, t-1\} \backslash \{j\}$ and $\deg(U_j(X)) = t-1$.
Then we set:

$$D(X) = c_0 + c_1 X + \cdots + c_{d-1} X^{d-1} + \textstyle\sum_{j=1}^{t} U_j(X)(c_d X + \cdots + c_{2d-t-1} X^{d-t})$$
$$+ \textstyle\sum_{j=1}^{t} U_j(X) \sum_{i=1}^{t-1} c_{2d-t-1+i} u_j^{d-t+i}.$$

The degree $d-1$ polynomial $D(X)$ satisfies $D(u_i) = C(u_i) = x_i x'_i$ of $i \in \{0, \ldots, t-1\}$.
In order to build efficiently $\texttt{DFT}_\omega(D(X))$, let's write:

$$D(X) = c_0 + c_1 X + \cdots + c_{d-1} X^{d-1} + (c_d X + \cdots + c_{2d-t-1} X^{d-t}) \textstyle\sum_{j=1}^{t} U_j(X)$$
$$+ \textstyle\sum_{i=1}^{t-1} c_{2d-t-1+i} \sum_{j=1}^{t} U_j(X) u_j^{d-t+i}$$

Thus:

$$\begin{aligned} \texttt{DFT}_\omega(D(X)) = \; & \texttt{DFT}_\omega(C(X)) \\ & - \texttt{DFT}_\omega(c_d X^d + \cdots + c_{2d-2} X^{2d-2}) \\ & + \texttt{DFT}_\omega(c_d X + \cdots + c_{2d-t-1} X^{d-t}) \odot \vec{u} \\ & + \textstyle\sum_{i=1}^{t-1} c_{2d-t-1+i} \cdot G_i. \\ = \; & \texttt{Mask}(\vec{x} \odot \vec{x}') \; . \end{aligned}$$

where: $G_i = \texttt{DFT}_\omega(\sum_{j=1}^{t} U_j(X) u_j^{d-t+i})$ for $i \in \{1, \ldots, t-1\}$ and $\vec{u} = \texttt{DFT}_\omega(\sum_{j=1}^{t} U_j(X))$ are a precomputed values, and $c_d, \ldots, c_{2d-2} = \texttt{extractLastCoefficients}(\vec{z} \odot \vec{z}\,')$. We remind that *extractLastCoefficients* has been defined in [10]:

We have seen that $\texttt{IDFT}_\omega(\vec{z} \odot \vec{z}\,') = (c_i)_{i \in \{0, \ldots, n-1\}} = C(X)$, then if we denote $\vec{y} = \vec{z} \odot \vec{z}\,'$, by definition $c_{j+d} = \sum_{i=0}^{n-1} y_i \omega^{-i(j+d)} = \sum_{i=0}^{n-1} (y_i \omega^{-id}) \omega^{-ij} \; \forall \; 0 \leq j \leq d-1$ and $(c_{j+d})_{j \in \{0, \ldots, d-1\}}$ is obtained from $\texttt{IDFT}\left((y_i \omega^{-id})_{0 \leq i \leq n-1}\right)$.

If we denote $\phi(C, \omega) = -\texttt{DFT}_\omega(c_{d+1} X^d + \cdots + c_{2d-2} X^{2d-2}) + \texttt{DFT}_\omega(c_d X + \cdots + c_{2d-t-1} X^{d-t}) \odot \vec{u} + \sum_{i=1}^{t-1} c_{2d-t-1+i} \cdot G_i$ where C represents the $d-1$ last coefficients of $\texttt{IDFT}(\texttt{Mask}(\vec{x}) \odot \texttt{Mask}(\vec{x}'))$, then we get that

$$\texttt{Mask}(\vec{x} \odot \vec{x}') = \texttt{Mask}(\vec{x}) \odot \texttt{Mask}(\vec{x}') + \phi(C, \omega)$$

3.5 Security Proof

We propose to show in this section that our method corresponds to $(d-t)$-probing order for the security with a discussion around more sophisticate security models. For fault injection resilience, we assume that we are in the random fault model with a reasonable number of injected faults.

SCA Resilience We showed that this construction is identical to the original construction of [10] up to the sign and up to the parity of n. The proof is coming from the property of this masking that can be written as a DSM encoding [8]:

$$x \mapsto (x, r) \mapsto xG + rH.$$

For this model, the masking order is provided by the minimal distance of the code $H^\perp$. It is proven in [10] that the probing order depends of code $H^\perp$ which can be MDS (i.e $d_{min} = n-(d-t)+1$) or AMDS (i.e $d_{min} = n-(d-t)$). We show in the subsection 4 (with $t=1$) that we are in the MDS favourable case. It means that odm equals $d-t$. The gadget multiplication is also $(d-t)$-probing secured due to the MDS property of H.

For more sophisticate security models like `IOS`, `SNI` or t-region probing, we refer to [18] for a similar method involving the DFT computation and [30] which gives proves for a code based masking. According to the proof of [10], the masking method of this paper satisfies the main properties of [18,30]. However proving rigorously that the presented masking method is $(d-t)$-`SNI` is out of the scope of this paper and will be covered in another one.

Faults Injection By construction, everywhere a codeword C is present, the integrity of C can be checked by computing the syndrome of C, i.e by computing $\texttt{IDFT}(C) = c$ and checking that c corresponds to a degree $d-1$ polynomial. If not, it means that some errors have been introduced. According our parameters, C belongs to the Reed-Solomon code $\mathrm{RS}[2d, d, d+1]$ and can detect d errors.

The difficult question concerns the gadget multiplication between two vectors $\texttt{Mask}(x)$ and $\texttt{Mask}(y)$. For this computation we must perform $\texttt{Mask}(x) \odot \texttt{Mask}(y)$ where "$\odot$" corresponds to the multiplication term by term. We showed that $\texttt{Mask}(x) \odot \texttt{Mask}(y) = \texttt{DFT}_\omega(C(X))$ where $C(X)$ is a degree $2d-2$ polynomial. However, our codewords have length $n = 2d$ and $\texttt{DFT}_\omega(C(X)) \in \mathrm{RS}[2d, 2d-1, 2]$. Hence we can check with a syndrome calculation (i.e $\texttt{IDFT}(\texttt{Mask}(x) \odot \texttt{Mask}(y))$) that $C(X)$ is degree $2d-2$ polynomial. If not, it means that at least one error has been injected. Then an attacker may inject faults on the vector $(c_d, \ldots, c_{2d-2})$, however in this case we remind that

$$\texttt{Mask}(\vec{x} \odot \vec{y}) = \texttt{Mask}(\vec{x}) \odot \texttt{Mask}(\vec{y}) + \phi((c_d, \ldots, c_{2d-2}), \omega),$$

with $\texttt{Mask}(\vec{x} \odot \vec{y})$ and $\texttt{Mask}(\vec{x}) \odot \texttt{Mask}(\vec{y})$ that can be verified, then the injected fault will be detected.

Remark 8. We assume that fault are randoms and do not directly affect the syndrome computations. We showed here that our gadget supports one fault injection. To support more injections we could modify our encoder by reducing the dimension of $\vec{r}$. As a direct consequence, the degree of the resulting polynomial $C(X)$ from a multiplication has a degree strictly less than $2d-2$ and more errors can be detected. In the same time this modification decrease the security probing order, thus, it is now a question of balance.

3.6 Complexity

It is shown in [10] that the complexity of the multiplication is quasi-linear and requires $\mathcal{O}(4d\log(2d))$ multiplications in $\mathbb{F}_q$. This is a standard complexity, but regarding real performances and applicability a study must be performed over different platforms (hardware and software) with different strategies: parallel computation, pipeline, bitslicing... From now on, we set $t = 1$ as it seems us difficult to take benefit of several symbol encoding due to the design of CRYSTALS-Kyber. Taking $t > 1$ may be interesting if we manage to compute simultaneously several KEM computation but it affects the probing security order.

In terms of randomness, we require $d - t$ random symbols to mask t sensitive ones. As the multiplication includes a refresh done by adding the mask of $\vec{0}$, it requires another batch of $d - t$ random symbols.

3.7 Masking a Polynomial Function

By induction, we can compute $\texttt{Mask}(x^n)$ for an arbitrary n value in $\mathbb{Z}$. We can write $\texttt{Mask}(x^n) = \texttt{Mask}(x^{n-1} * x) = \texttt{Mask}(x) \odot \texttt{Mask}(x^{n-1}) + \phi(C, \omega)$ thus if we assume that $\texttt{Mask}(x^{n-1})$ has been computed, then the property is demonstrated. The same proof holds for Horner (polynomial evaluation) algorithm.

3.8 Masking a Formal Polynomial

Let $s(X), u(X) \in (\mathbb{F}_q^{(deg)}[X])^2$. We define

$$\texttt{Mask}(s(X)) = \sum_{i=0}^{deg} \texttt{Mask}(s_i) X^i \tag{6}$$

Also, we have $\texttt{Mask}(s(X)) \odot \texttt{Mask}(u(X)) = \texttt{Mask}(s(X) * u(X))$. We deduce that we can perform $\texttt{Mask}(s(X) * u(X))$ using the Karatsuba algorithm of complexity $deg^{1.585}$ $\texttt{Mask}$ multiplications [21].
We note that fast Fourier transformed based on Cooley-Turkey algorithm that involves a n-root of unity could be obviously applied since the scalar multiplication is well defined over the linear codes. We preferred Karatsuba because this part of CRYSTALS-Kyber algorithm is not the most costly in term of masking.

3.9 Masking Boolean Operations

We proved that we can use our masking method whenever arithmetical operations are used. However, some algorithms requires boolean operations. While we cannot tweak our masking method to work for every value, lets recall some simple properties working for $x, y \in \{0, 1\}^2$:

$$x \wedge y = x * y, \;\; x \oplus y = x + y - 2 * x * y$$

Thus, in the very specific case where the value before masking is equal to 0 or 1, we are able to perform basic boolean operations while in masked state by using these arithmetic equations.

3.10 Masking Keccak

The current standard for hash functions is FIPS-202 [13], also known as Keccak or SHA3. It can be seen as a 3-dimensional array denoted by A of size $(5, 5, w)$. At its core is the $\texttt{Round}$ function and its 5 steps:

- θ: The first step uses three substeps:
  1. $\forall(x,z) \in [\![0,4]\!] \times [\![0,w-1]\!], C[x,z] = \bigoplus_{i=0}^{4} A[x,i,z]$
  2. $\forall(x,z) \in [\![0,4]\!] \times [\![0,w-1]\!], D[x,z] = C[(x-1) \bmod 5, z] \oplus C[(x+1) \bmod 5, (z-1) \bmod w]$
  3. $\forall(x,y,z) \in [\![0,4]\!]^2 \times [\![0,w-1]\!], A'[x,y,z] = A[x,y,z] \oplus D[x,z]$

  All the operations here are relying on XOR ($\oplus$) and manipulates bits. Thus, our masking method can be directly applied.
- ρ: The second step simply rotates the elements of the array on the z axis. We won't detail it here as it is obviously with our masking method.
- π: Similarly to the ρ step, this step rotates the positions of the lanes (z axis) in the array. As for the ρ step, we won't detail it. It is fully compatible with our masking method.
- χ: This step is the non-linear step of Keccak. It computes the following:
  $$\forall(x,y,z) \in [\![0,4]\!]^2 \times [\![0,w-1]\!],$$
  $$A'[x,y,z] = A[x,y,z] \oplus ((A[(x+1) \bmod 5, y, z] \oplus 1) \wedge A[(x+2) \bmod 5, y, z])$$
  As we are also able to mask the AND ($\wedge$) operation, our method can be applied to this step as well.
- ι: The final step involves only one lane. It is XORed with a round constant, and thus can be easily masked:
  $$\forall z \in [\![0,w-1]\!], A'[0,0,z] = A[0,0,z] \oplus RC[z]$$
  With this final step covered, we can see that our method can be applied to every step in the `Round` function of Keccak.

Keccak also uses the `Sponge` protocol to compute the hash of a word. As the two part of the protocol, `Absorb` and `Squeeze`, simply involves XORing the word with the on-going state and extracting a copy of the state, we won't discuss them in detail as their compatibility with our masking method is once again obvious. For more details on the Keccak function, please refer to the standard paper from the NIST [13].

4 CRYSTALS-Kyber Example

4.1 Discussion on Parameters

As stated in Section 2.2, CRYSTALS-Kyber operations are defined over $\mathbb{Z}_q$ with $q = 3329$ satisfying $q - 1 = 2^8 \times 13$. If $\nu = 3$ is a primitive element of $\mathbb{F}_q$, we could set $n = 13 = 2d' + 1$, $d = 6$, with $\omega = \nu^{\frac{q-1}{13}}$ and $\alpha = \nu^{\frac{q-1}{16}}$. The masking method is working for these parameters but we have not found a better way to compute the `DFT` than using a Vandermonde matrix multiplication which costs $\mathcal{O}(n^2)$. However, with these parameters, for $t = 1$, we get $odm = d + 1 - 1 = 6$ and 6 faults can be detected on codewords.

Another choice can be: $n = 8 = 2d'$, $d = 4$, $\omega = \nu^{\frac{q-1}{8}} = 749$ and $\alpha = \nu^{\frac{q-1}{13}} = 2970$. We chose to these parameters,

$$A = \begin{pmatrix} 1 & \alpha & \alpha^2 & \alpha^3 \\ 1 & \alpha^2 & \alpha^4 & \alpha^6 \\ 1 & \alpha^3 & \alpha^6 & \alpha^9 \\ 1 & \alpha^4 & \alpha^8 & \alpha^{12} \end{pmatrix}, \quad V(\omega) = \begin{pmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & \omega & \omega^2 & \omega^3 & \omega^4 & \omega^5 & \omega^6 & \omega^7 \\ 1 & \omega^2 & \omega^4 & \omega^6 & 1 & \omega^2 & \omega^4 & \omega^6 \\ 1 & \omega^3 & \omega^6 & \omega & \omega^4 & \omega^7 & \omega^2 & \omega^5 \end{pmatrix}$$

Then,

$$\texttt{Mask}(x) = (x, r_1, r_2, r_3) \times (A^{-1})^\top \times V(\omega),$$

and

$$\texttt{Mask}(x) = (x, r_1, r_2, r_3) \times \begin{pmatrix} 3106 & 2858 & 3174 & 2115 & 2750 & 487 & 2997 & 3238 \\ 420 & 3028 & 750 & 2228 & 758 & 2346 & 2288 & 3272 \\ 2568 & 1881 & 1558 & 1208 & 1834 & 526 & 338 & 2683 \\ 565 & 2221 & 1177 & 1108 & 1317 & 3300 & 1036 & 795 \end{pmatrix},$$

with $\vec{r} = (r_1, r_2, r_3) \in \mathbb{Z}_q$ is picken randomly.

$$\texttt{Mask}(x) = xG + \vec{r}H, \text{ with } H = \begin{pmatrix} 420 & 3028 & 750 & 2228 & 758 & 2346 & 2288 & 3272 \\ 2568 & 1881 & 1558 & 1208 & 1834 & 526 & 338 & 2683 \\ 565 & 2221 & 1177 & 1108 & 1317 & 3300 & 1036 & 795 \end{pmatrix}.$$

It is possible to check (cf: MAGMA online) that the minimal distance of $H^\perp$ is 4 as predicted by the theory, then $odm = 3$. Furthermore $V(\omega)$ is a Reed Solomon code and 4 faults can be detected. The complexity of the detection corresponds to complexity of the syndrome computation that can be achieved with a DFT and for $n = 8$ we have the following tree decomposition:

$$\begin{tikzcd}
 & & & X^8-1 \arrow[dll] \arrow[drr] & & & & \\
 & X^4-1 \arrow[dl] \arrow[dr] & & & & X^4+1 \arrow[dl] \arrow[dr] & & \\
X^2-1 \arrow[d] \arrow[dr] & & X^2+1 \arrow[d] \arrow[dr] & & X^2+2620*X+1 \arrow[d] \arrow[dr] & & X^2+709*X+1 \arrow[d] \arrow[dr] & \\
X-1 \arrow[d] & X+1 \arrow[d] & X+1600 \arrow[d] & X+1729 \arrow[d] & X+40 \arrow[d] & X+2580 \arrow[d] & X+749 \arrow[d] & X+3289 \arrow[d] \\
P(1) & P(-1) & P(-1600) & P(-1729) & P(-40) & P(-2580) & P(-749) & P(-3289)
\end{tikzcd}$$

It is shown in [31] that this representation is favourable to hardware implementation and complexity does not exceed $nlog(n)$ multiplications over $\mathbb{F}_q$.

We also propose to consider the parameters $n = 4 = 2d'$, $d = 2$, $\omega = \nu^{\frac{q-1}{4}} = 1729$ and $\alpha = \nu^{\frac{q-1}{13}} = 2970$. We chose to these parameters,

$$A = \begin{pmatrix} 1 & \alpha \\ 1 & \alpha^2 \end{pmatrix}, \ V(\omega) = \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & \omega & \omega^2 & \omega^3 \end{pmatrix}$$

Then,

$$\texttt{Mask}(x) = (x, r) \times (A^{-1})^\top \times V(\omega),$$

$$\texttt{Mask}(x) = (x, r) \begin{pmatrix} 103 & 2590 & 1545 & 2387 \\ 3227 & 740 & 1785 & 943 \end{pmatrix} = xG + rH,$$

which leads to a minimal distance of $H^\perp$ equal to 2 and consequently, $odm = 1$. These parameters lead also to a very fast DFT with the following tree decomposition:

$$\begin{tikzcd}
& & & X^4-1 \arrow[dll] \arrow[drr] & & & \\
& X^2-1 \arrow[dl] \arrow[dr] & & & & X^2+1 \arrow[dl] \arrow[dr] & \\
X-1 \arrow[d] & & X+1 \arrow[d] & & X+1600 \arrow[d] & & X+1729 \arrow[d] \\
P(1) & & P(-1) & & P(-1600) & & P(-1729)
\end{tikzcd}$$

4.2 Masking Strategy

First we will focus on securing the PKE Decapsulation part of CRYSTALS-Kyber (see [2], Algorithm 6 page 9 or Algorithm 5 in Appendix B). Then we will discuss how we can extend our masking method to the entirety of the KEM Decapsulation procedure (see Algorithm 1) and CRYSTALS-Kyber itself.
You can use the following map (Figure 1) of the KEM Decapsulation to navigate between the different parts we had to mask.

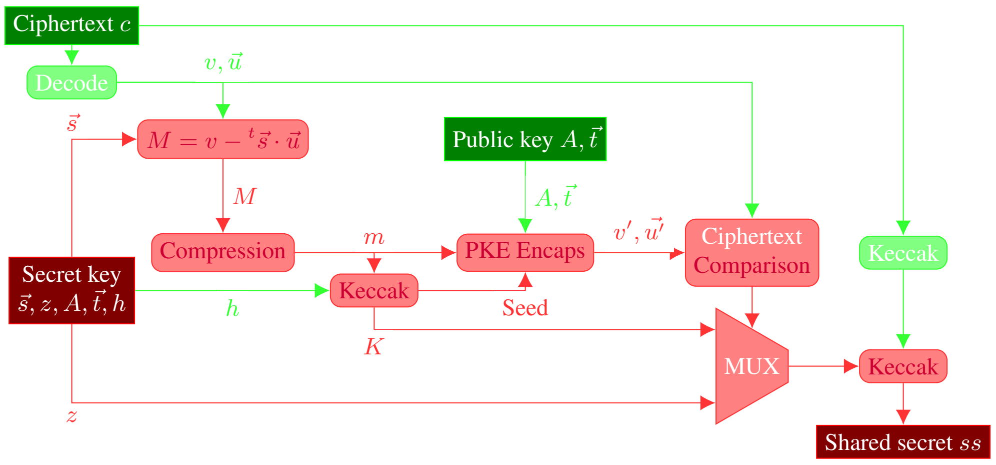

Fig. 1. Interactive map (links) of our masking strategy

Graph legend:

- 🟢 : Non-sensitive operation
- 🟩 : Non-sensitive input/output of the algorithm
- ⟶ : Non-sensitive intermediate data
- 🔴 : Sensitive operation
- 🟥 : Sensitive input/output of the algorithm
- ⟶ : Sensitive intermediate data

We propose to mask the following operation: $v - {}^t\vec{s} \cdot \vec{u}$ with $v \in R_q$ given, $\vec{u} \in R_q^{sec}$ given, and $\vec{s} \in R_q^{sec}$ the secret. First, we have to discuss how to multiply a sensitive data and a public one.

Karatsuba between a Sensitive and a Public Polynomials To multiply two sensitive polynomials we rely on the Karatsuba algorithm. However, to avoid the cost of masking a public polynomial, we will instead consider its coefficients as scalars. Indeed, we have the following theorem:

Theorem 1. *Let $f(X) \in R_q$ be a sensitive data and $g(X) \in R_q$ a public one. Then $\texttt{Mask}(f(X) * g(X)) = \texttt{Mask}(f(X)) * g(X)$*

Proof. First we will prove it for a degree lesser than 16 and then recursively extend the theorem to the entirety of the Karatsuba algorithm as for this degree, we use the textbook polynomial. The textbook polynomial multiplication states that

$$\left(\sum_{i=0}^{15} f_i X^i\right) * \sum_{j=0}^{15} g_j X^j = \sum_{j=0}^{15} g_j * \left(\sum_{i=0}^{15} f_i X^{i+j}\right) \tag{7}$$

Using the linearity of our masking method, it's immediate that

$$\texttt{Mask}(\sum_{j=0}^{15} g_j * \left(\sum_{i=0}^{15} f_i X^{i+j}\right)) = \sum_{j=0}^{15} g_j * \left(\sum_{i=0}^{15} \texttt{Mask}(f_i) X^{i+j}\right) \tag{8}$$

Now, let's recall the Karatsuba formula:

$$\begin{aligned} f(X) * g(X) &= (f'(X) + f''(X)X^{n/2}) * (g'(X) + g''(X)X^{n/2}) \\ &= f' * g'X^n + (f' * g' + f'' * g'' + (f' + f'') * (g' + g''))X^{n/2} + f'' * g'' \end{aligned}$$

Each product is once again between a sensitive polynomial f and a public one g. Thus, we have to prove that $\texttt{Mask}(f'(X)*g'(X)) = \texttt{Mask}(f'(X))*g'(X)$(same for $f''*g''$ and $(f'+f'')*(g'+g'')$) at degree k. If true, then using the linearity it'll also be true for $f(X)*g(X)$ at degree $2k$. However, we already proved it for degree lesser than 16 in Equation 8. By recursion we proved that we do not need to mask nonsensitive data before multiplying them with sensitive polynomials as we can see their coefficients as scalars.

Thanks to the Theorem 1 and the homomorphic properties of the `Mask` procedure we have:

$$\texttt{Mask}(v - {}^t\vec{s} \cdot \vec{u}) = \texttt{Mask}(v) - \sum_{i=0}^{sec} \texttt{Mask}(s_i) * u_i,$$

where ${}^t\vec{s} \cdot \vec{u} = \sum_{i=0}^{sec}(s_i * u_i)$ with $s_i \in R_q$ and $u_i \in R_q$.

Compression Following the CRYSTALS-Kyber design described in Algorithm 5, we have to apply the $Compress_q$ function while staying masked. We have the following theorem:

Theorem 2. *$Compress_q$ can be computed using a polynomial function.*

Proof. We can rewrite the $Compress_q$ function from Equation 1 as

$$\forall \alpha \in \mathbb{Z}_q,\ Compress_q(\alpha) = \begin{cases} 1 & \text{if } \lceil \frac{q}{4} \rfloor < \alpha < \lceil \frac{3q}{4} \rfloor \\ 0 & \text{otherwise} \end{cases} \tag{9}$$

As $\mathbb{Z}_q$ is a finite field, we can simply enumerate all the values of α resulting in 1 and those resulting in 0 and thus we can rewrite Equation 9 as

$$\forall \alpha \in \mathbb{Z}_q,\ Compress_q(\alpha) = \begin{cases} 1 & \text{if } \alpha \in \{\alpha_0, \alpha_1, \dots, \alpha_h\} \\ 0 & \text{if } \alpha \in \{\beta_0, \beta_1, \dots, \beta_l\} \end{cases} \text{ with } h + l = q \tag{10}$$

A simple Lagrange interpolation of Equation 10 thus give us the following:

$$\forall \alpha \in \mathbb{Z}_q,\ Compress_q(\alpha) \Leftrightarrow P(X = \alpha) = \prod_{i=0}^{l}(X - \beta_i) \sum_{j=0}^{h} \prod_{k=0, k \neq j}^{h} \frac{(X - \alpha_k)}{(\alpha_j - \alpha_k)} \tag{11}$$

We proved $Compress_q$ can be seen as a polynomial function and we can further extend the reach of our masking method within CRYSTALS-Kyber.

We deduce from this theorem that we can mask $Compress_q$ as a polynomial function. It requires q `Mask` multiplications and q `Mask` additions by using the Horner algorithm. However, we can improve this complexity.

In the specific case of our $Compress_q$ interpolation, we notice that the polynomial function has a structure. All the exponents are odd. Thus, a first optimization is to change the variable from X to $Y = X^2$, thus reducing the complexity from q `Mask` multiplications to $q/2$ `Mask` multiplications. Then, we notice we can factorize that polynomial into 838 polynomials, with the largest degree being 599. Thus, by multiplying some of these polynomials together, we end up with 2 polynomials of degree 599 and one of degree 466. By precomputing the Y^i, and not masking the polynomial coefficients[5] and using them as scalars as described in 3.3, we reduce the complexity (multiplications wise) of our masked $Compress_q$ to just around 600 `Mask` multiplications.

From PKE to KEM PKE Decaps is used in KEM Decapsulation as shown in Algorithm 1. In order to mask the rest of the KEM Decapsulation procedure of Kyber and the other KEM procedures, we have to address a few points.

- **How do we hash the message output of PKE Decaps?** As stated in Subsection 3.10, we can only use our Keccak implementation if the value masked is either 0 or 1. Which is the case for each term of the output of the $Compress_q$ function. Thus, we can directly apply our Keccak implementation on the output of PKE Decaps, albeit if this output is kept in polynomial form and masked.
- **How do we mask PKE Encapsulation?** The homomorphic properties of `Mask` can also be applied to both the PKE encapsulation and key generation of CRYSTALS-Kyber. By taking into account Remark 3 regarding the $Decompress_q$ function, we can secure most of the computations using the sensitive data $\vec{s}, m, \vec{r}, \vec{e}, \vec{e_1}$ and e_2 in PKE Key Gen (see [2] page 8 Algorithm 4 or Algorithm 3 Appendix B) and PKE Encaps (see [2] page 9 Algorithm 5 or Algorithm 4 Appendix B). For instance, to compute v we do

$$\texttt{Mask}(v) = \left(\sum_{i=0}^{sec} t_i * \texttt{Mask}(r_i) \right) + \texttt{Mask}(e_2) + 1665 * \texttt{Mask}(m) \tag{12}$$

  However, we have to secure the sampling of these sensitive data.

[5] Those are not sensitive data

- **How do we use our masking method to perform sensitive data sampling?** To sample sensitive values in the PKE Encapsulation procedure, we use the CBD from Equation 4 fed by two chained Keccak instances. As the output of our masked Keccak implementation is a vector of either 0s or 1s masked, it can be fed into a new masked Keccak implementation without problems.
  We can compute the following:

$$\texttt{Mask}(CBD_\eta(G)) = \sum_{i=0}^{255}(\sum_{j=0}^{\eta}\texttt{Mask}(G_{2i\eta+j}) - \sum_{j=0}^{\eta}\texttt{Mask}(G_{2i\eta+\eta+j}))X^i \tag{13}$$

  The result will be masked with our method and ready for use. We can perform sensitive data sampling from a masked seed and remain in the masked domain all along.

  Remark 9. Some points discussed here also apply to the sampling of the message in the KEM Encapsulation procedure. Using a TRNG and masking its output, we can have a masked message from the start and thus compute the seed used in the CBD while always staying masked.

- **How do we compare ciphertexts in the Fujisaki-Okamoto Transform without unmasking them?** One of the biggest issue with masking Kyber is the lossy nature of the $Compress_q$ function, as stated in Remark 4. As the ciphertext in the KEM Decapsulation (Algorithm 1) is given as input in a compressed state, the reference paper [6] simply compresses the generated ciphertext into c' and compares it with the input ciphertext c.
  However, we have already seen that masking the $Compress_q$ function can be costly. Thus, papers masking Kyber [7,9] use a different approach: They compare the generated ciphertexts $\vec{u'}, v'$ with the decompression of c.
  We went a step further and relied on the property stated in Remark 4 Equation 3. A key point here is we want a function that returns 0 when the ciphertexts are good and not 0 when the comparison fails. Which means that, instead of performing a Lagrange interpolation[6] like for the message compression, here we can just list the values of $y = x - x'$ such as $|y| \leq \lceil q/2^{d_i+1} \rfloor \bmod q$ and consider them as the roots of the polynomial we are looking for. Thus, for $d_i = d_u = 10$, we have $\lceil q/2^{d_i+1} \rfloor = \lceil q/2^{10+1} \rfloor = 2$, thus $y \in [\![-2, 2]\!]$, 5 roots and a polynomial of degree 5:

$$P(X) = X^5 + 3324 * X^3 + 4 * X = X * (Y-4) * (Y-1) \text{ with } Y = X^2 \tag{14}$$

  This Equation 14 only requires 3 masked multiplications. For $d_i = d_v = 4$, we have $\lceil q/2^{d_i+1} \rfloor = \lceil q/2^{4+1} \rfloor = 104$, thus $y \in [\![-104, 104]\!]$, 209 roots and a polynomial of degree 209. However, we know that X will be a factor of this polynomial and that we will be able to use the symmetric nature of the set of roots to have $(X-a)*(X+a) = X^2 - a^2$, thus allowing us to have two polynomials, X and one of degree 104 in $Y = X^2$. A last optimization is to factorize together some of the factors of this polynomial in Y to have 8 factors of degree 13, requiring only $1+1+7+12 = 21$ masked multiplications[7].

[6] All the mathematical optimizations in this paper were computed using PARI GP.

[7] For 4 factors of degree 26, 30 masked multiplications; for 16 factors of degrees 6 and 7, 23 masked multiplications

We demonstrated that our masking method can be applied completely to CRYSTALS-Kyber[8] to secure computations on sensitive data, without requiring any conversion to a different masking method and providing an error detection and error correcting capability.

5 Implementation and Performances

We made a *proof of concept* implementation of our masking method in the C language on a desktop. This first implementation may not be the most optimized but we think it is important to share some performances[9] results to give an idea of the costs of our method. Please note that, to the best of our knowledge, there is no paper discussing performances of a masked implementation of CRYSTALS-Kyber on desktop we could compare to. Thus, we will only present our results here and won't be able to compare them with other methods for now.

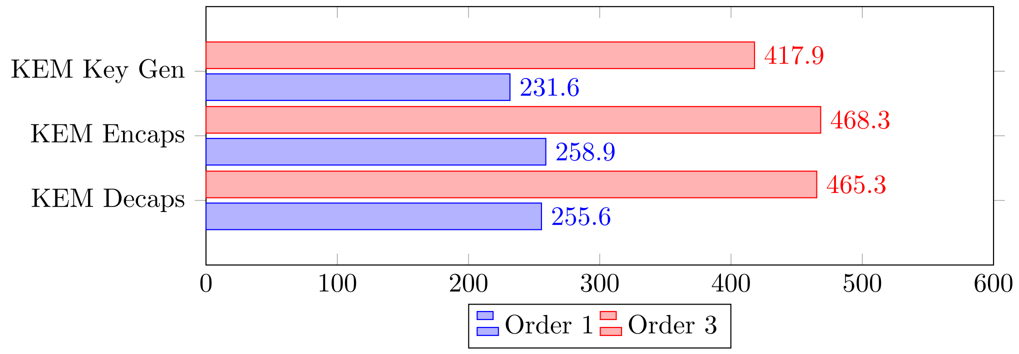

Fig. 2. Performances in milliseconds of our masked KYBER512

It's interesting to notice on Figure 2 that we double the performances while tripling the masking order. The reason behind it is that we have to double the length of the Reed-Solomon code used to perform the masking. This tendency is also noticeable in the following Table 1:

Table 1. Performances in milliseconds of several important functions

Masking order	1	3
Masking a degree 256 polynomial	0.007	0.0136
Ciphertexts comparison	1.07	1.93
Karatsuba between a sensitive and a public polynomials	0.717	1.358
Karatsuba between two sensitive polynomials	2.46	4.53
Message compression	19.6	35.1
Hash function	25.2	46.8

[8] Note that we mask KYBER512 but our method works for other security levels as well.
[9] Averages in milliseconds over 1000 iterations.

The most costly functions we have to use are our Hash function and the message compression, as seen in Table 1. They take the vast majority of the run time of our implementation, as seen in the following Figure 3:

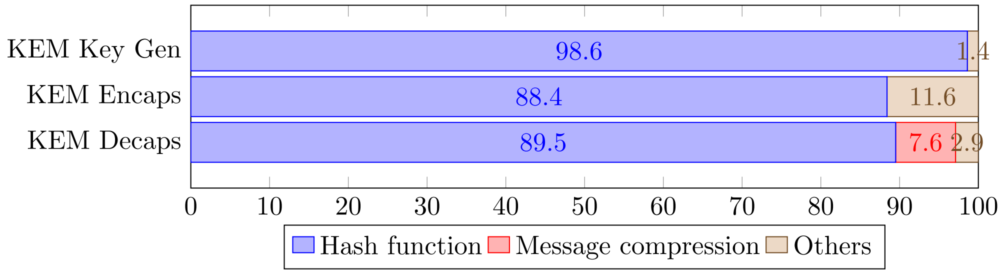

Fig. 3. Percent of run time of functions at order 3

We also compared these results with the reference C source code of CRYSTALS-Kyber:

- KEM Key Gen: from 0.03ms to 231.6ms and 417.9ms ($\times$7720 and $\times$13 930)
- KEM Encapsulation: from 0.03ms to 258.9ms and 468.3ms ($\times$8630 and $\times$15 610)
- KEM Decapsulation: from 0.03ms to 255.6ms and 465.3ms ($\times$8520 and $\times$15 510)
- Keccak: from 0.0007ms to 25.2ms and 46.8ms ($\times$36 000 and $\times$66 857)

Remark 10. Note that our implementation is not fully optimized while the reference C source code of CRYSTALS-Kyber is. The interesting result here is the massive overhead in the specific case of Keccak, stressing the need for alternatives solutions to our masked implementation of SHA3.

The performances shown here were realized on a DELL Precision 3561 laptop equipped with a 11th Gen Intel(R) Core(TM) i7-11850H processor operating at 2.50 GHz, 16 GB of RAM. The source code was compiled and executed using gcc version 11.3.0. A particularity of our setup is the use of Ubuntu 22.04.1 through WSL2 (Windows Subsystems for Linux) on a computer operating Windows 11.

We plan on making our source code public using an anonymized Github depot.

5.1 Possible Improvements

In this subsection, we propose some ideas to improve the performances of the implementation albeit at the expanse of some of our masking method advantages.

- The most costly function in our CRYSTALS-Kyber is our Keccak implementation, as shown by the Figure 3 and Remark 10. We propose the idea of replacing it by a boolean masked implementation of Keccak, like [5] from Bertoni et al. for instance. To convert the masked output of this implementation to our code-based masking, one simply needs to use our method on each boolean share and then use our XOR to retrieve the value which was boolean masked but now in our code-based masking domain. However, giving a code-based

mask data to the Bertoni's Keccak implementation is a bit more tricky.
Let $m \in R_q$ be the sensitive data we masked in the code-based making domain. We have $\texttt{Mask}(m) = (\texttt{Mask}(m_i))_{i \in [\![1..256]\!]}$. Let's pick randomly $(R_1, \ldots, R_{d'}) \in \left(\mathbb{Z}_q^{256}\right)^{d'}$. Then, we can compute $\texttt{Mask}(m) + \sum_{i=1}^{d'} \texttt{Mask}(R_i) = \texttt{Mask}(m + \sum_{i=1}^{d'} R_i)$. Finally we get $m + \sum_{i=1}^{d'} R_i = \texttt{Unmask}(\texttt{Mask}(m) + \sum_{i=1}^{d'} \texttt{Mask}(R_i))$.
We thus have an arithmetic masking of the m, and we can apply methods like Goubin `A2B` [16] or the more recent `SecA2BModp` from Bronchain-Cassiers [9] to switch to boolean masking compatible with Bertoni's masked Keccak.

– We already have made some optimizations to our $Compress_q$ function. However, it still is the second most costly function in our CRYSTALS-Kyber implementation. Thus, we propose the idea of converting from our code-based masking to an arithmetic one to use already existing work from Bos et al. or Bronchain and Cassiers [7,9] to mask this function at a lower cost. The conversion process is the same as for the previous optimization proposal aimed at Keccak, without the `A2B` conversion requirement.

Remark 11. Please note that we focused on the mathematical and algorithmic optimizations throughout this paper, there might also be software optimization for our source code to improve its performances. Also note that these performances were obtained on a desktop implementation and thus might be far better once our masking is adapted to a hardware or codesign implementation.

6 Conclusion

In this paper we demonstrated the first masked implementation of a post-quantum KEM using the code-based masking method in Section 4. We proved in Section 3 that code-based masking can be used with finite fields of prime characteristic other than 2 and with code of even length. We studied how to mask a Keccak implementation in Section 3.10, however we leave the optimization of this implementation for another paper. We succeeded in proposing a masked implementation of CRYSTALS-Kyber where sensitive data are masked once and never require any conversion or unmasking. We also provide a better security against Fault Injection Attacks (FIA) by not only being able to detect faults but also to correct some.
The next step of our work is to implement this solution on a hardware platform and verify its robustness experimentally. We also plan to adapt our method to mask a post-quantum signature.

Acknowledgments This work was realized thanks to the grant 2022156 from the Appel à projets 2022 thèses AID Cifre-Défense by the Agence de l'Innovation de Défense (AID), Ministère des Armées (French Ministry of Defense). This paper is also part of the on-going work of Hensoldt SAS France for the Appel à projets Cryptographie Post-Quantique launched by Bpifrance for the Stratégie Nationale Cyber (France National Cyber Strategy) and Stratégie Nationale Quantique (France National Quantum Strategy). In this, Hensoldt SAS France is a part of the X7-PQC project in partnership with Secure-IC, Télécom Paris and Xlim.

A Graph legend

- ⬢ : Non-sensitive operation
- 🟩 : Non-sensitive input/output of the algorithm
- $\longrightarrow$: Non-sensitive intermediate data
- ⬢ : Sensitive operation
- 🟥 : Sensitive input/output of the algorithm
- $\longrightarrow$: Sensitive intermediate data

B CRYSTALS-Kyber

B.1 PKE

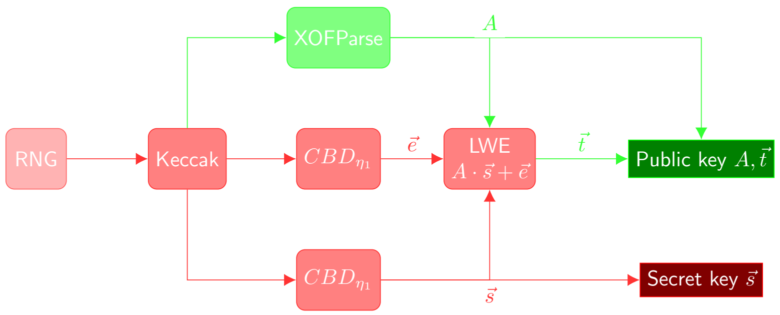

Fig. 4. Overview of the sensitive operations within the PKE Key generator

Algorithm 3 PKE-Kyber Key Generation

```
1: Output: Secret key $\vec{s} \in R_q^{sec}$
2: Output: Public Key $A \in R_q^{sec \times sec}, \vec{t} \in R_q^{sec}$
3: $A \leftharpoondown R_q^{sec \times sec}$
4: $\vec{s}, \vec{e} \hookleftarrow S_{\eta_1}^{sec} \times S_{\eta_1}^{sec}$
5: $\vec{t} := A \cdot \vec{s} + \vec{e}$
6: return $(\vec{s}, (A, \vec{t}))$
```

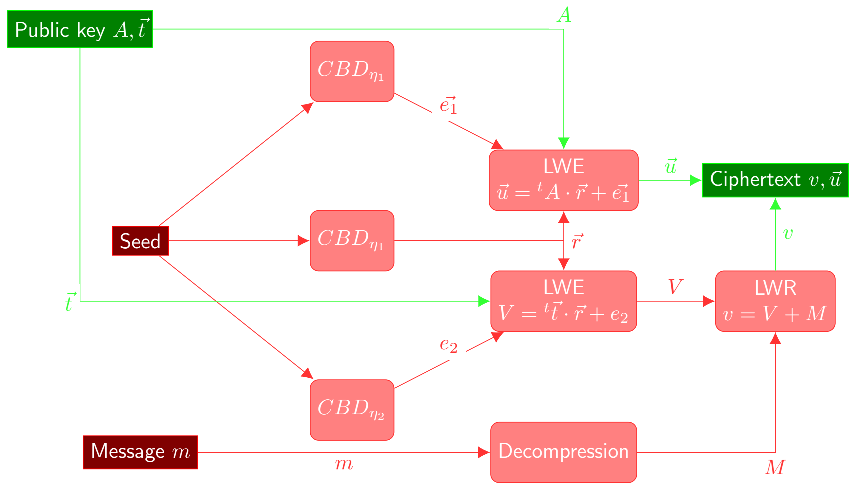

Fig. 5. Overview of the sensitive operations within the PKE Encapsulation

Algorithm 4 PKE-Kyber Encapsulation

```
1: Input: Public Key $A \in R_q^{sec \times sec}, \vec{t} \in R_q^{sec}$
2: Input: Message $m \in \{0,1\}^{256}$
3: Input: Seed $seed \in \{0,1\}^{256}$
4: Output: Ciphertext $\vec{u} \in R_q^{sec}, v \in R_q$
5: $\vec{r}, \vec{e_1}, e_2 \hookleftarrow S_{\eta_1}^{sec} \times S_{\eta_2}^{sec} \times S_{\eta_2}$
6: $\vec{u} := {}^tA \cdot \vec{r} + \vec{e_1}$
7: $v := {}^t\vec{t} \cdot \vec{r} + e_2 + \; Decompress_q(m)$
8: return $(\vec{u}, v)$
```

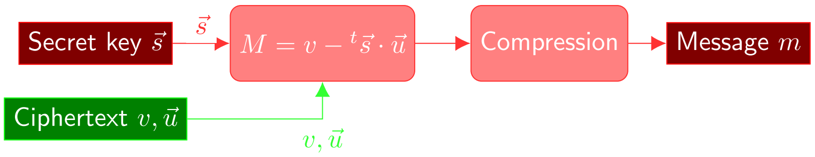

Fig. 6. Overview of the sensitive operations within the PKE Decapsulation

Algorithm 5 PKE-Kyber Decapsulation

```
1: Input: Secret key $\vec{s} \in R_q^{sec}$
2: Input: Ciphertext $\vec{u} \in R_q^{sec}, v \in R_q$
3: Output: Message $m \in \{0,1\}^{256}$
4: $m := Compress_q(v - {}^t\vec{s} \cdot \vec{u})$
5: return $m$
```

B.2 KEM

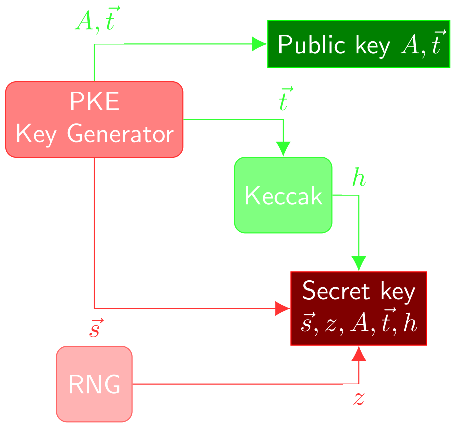

Fig. 7. Overview of the sensitive operations within the KEM Key generator

Algorithm 6 KEM-Kyber Key Generation

```
1: Output: Public Key $pk = (A, \vec{t})$
2: Output: Secret Key $sk = (\vec{s}, pk, H(pk), z \in \{0,1\}^{256})$
3: $z$ (256 random bits from system)
4: $\vec{s}, A, \vec{t} := PKE.KeyGen()$
5: $pk := (A, \vec{t})$
6: $sk := (\vec{s}, pk, H(pk), z)$
7: return $(pk, sk)$
```

Remark 12. Alternatively to store the matrix A as a part of the public key, we could store the random seed used to generate it in order to reduce the size of the public key.

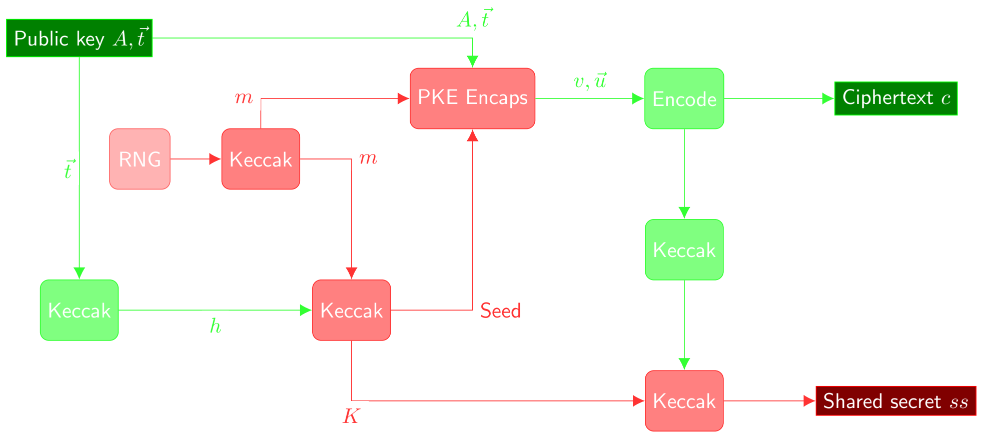

Fig. 8. Overview of the sensitive operations within the KEM Encapsulation

Algorithm 7 KEM-Kyber Encapsulation

```
1: **Input:** Public key $pk = (A, \vec{t})$
2: **Output:** Ciphertext $c$
3: **Output:** Shared secret $K \in \{0,1\}^{256}$
4: $m$ (256 random bits from system)
5: $m = H(m)$
6: $(K'', seed) := G(m \| H(pk))$
7: $\vec{u}, v := PKE.Encaps(pk, m, seed)$
8: $c = (Compress_q(\vec{u}, d_u), Compress_q(v, d_v))$
9: $K := KDF(K'' \| H(c))$
10: **return** $(c, K)$
```

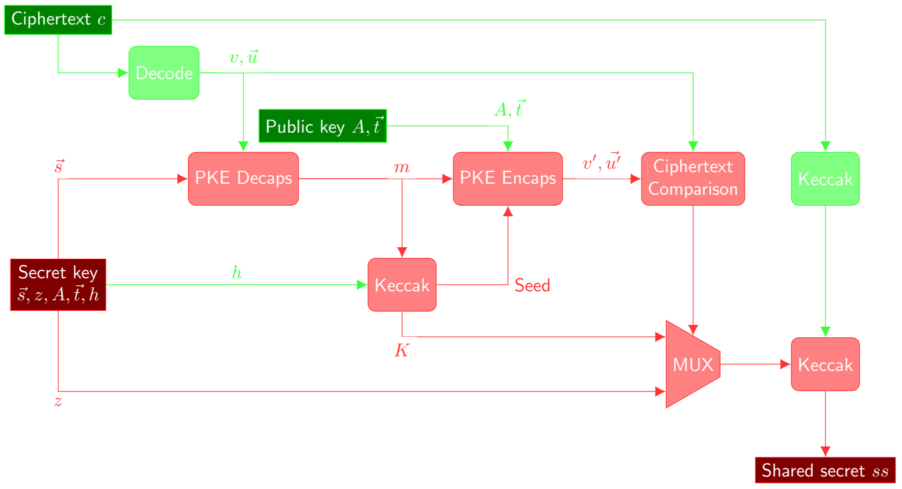

Fig. 9. Overview of the sensitive operations within the KEM Decapsulation

Algorithm 8 KEM-Kyber Decapsulation

```
1: **Input:** Ciphertext $c$
2: **Input:** Secret Key $sk = (\vec{s}, pk, H(pk), z)$
3: **Output:** Shared key $K$
4: $\vec{u}, v = Decompress_q(c, d_u), Decompress_q(c+, d_v)$
5: $m' := PKE.Decaps(\vec{s}, \vec{u}, v)$
6: $(K', seed') := G(m' \| H(pk))$
7: $\vec{u'}, v' := PKE.Encaps(pk, m', seed')$
8: $c' = (Compress_q(\vec{u'}, d_u), Compress_q(v', d_v))$
9: **if** $c == c'$ **then**
10:     **return** $KDF(K' \| H(c))$
11: **else**
12:     **return** $KDF(z \| H(c))$
13: **end if**
```

B.3 Parameters

Table 2. Parameter sets for CRYSTALS-Kyber

	NIST security level	n	q	k	η_1	η_2	d_u	d_v
KYBER512	I	256	3329	2	3	2	10	4
KYBER768	III	256	3329	3	2	2	10	4
KYBER1024	V	256	3329	4	2	2	11	5

References

1. Alagic, G., Apon, D., Cooper, D., Dang, Q., Dang, T., Kelsey, J., Lichtinger, J., Miller, C., Moody, D., Peralta, R., et al.: Status report on the third round of the nist post-quantum cryptography standardization process. US Department of Commerce, NIST (2022). https://doi.org/10.6028/NIST.IR.8413-upd1
2. Avanzi, R., Bos, J., Ducas, L., Kiltz, E., Lepoint, T., Lyubashevsky, V., Schanck, J.M., Schwabe, P., Seiler, G., Stehlé, D.: Crystals-kyber algorithm specifications and supporting documentation. pq-crystals (2021)
3. Backlund, L.: A side-channel attack on masked and shuffled implementations of m-lwe and m-lwr cryptography: A case study of kyber and saber (2023)
4. Beckwith, L., Abdulgadir, A., Azarderakhsh, R.: A flexible shared hardware accelerator for nist-recommended algorithms crystals-kyber and crystals-dilithium with sca protection. In: Cryptographers' Track at the RSA Conference. pp. 469–490. Springer (2023). https://doi.org/10.1007/978-3-031-30872-7_18
5. Bertoni, G., Daemen, J., Peeters, M., Van Assche, G.: Building power analysis resistant implementations of keccak. In: Second SHA-3 candidate conference. vol. 142. Citeseer (2010)
6. Bos, J., Ducas, L., Kiltz, E., Lepoint, T., Lyubashevsky, V., Schanck, J.M., Schwabe, P., Seiler, G., Stehlé, D.: 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). https://doi.org/10.1109/EuroSP.2018.00032
7. Bos, J.W., Gourjon, M., Renes, J., Schneider, T., Van Vredendaal, C.: Masking kyber: First-and higher-order implementations. IACR Transactions on Cryptographic Hardware and Embedded Systems pp. 173–214 (2021). https://doi.org/10.46586/tches.v2021.i4.173-214
8. Bringer, J., Carlet, C., Chabanne, H., Guilley, S., Maghrebi, H.: Orthogonal direct sum masking: A smartcard friendly computation paradigm in a code, with builtin protection against side-channel and fault attacks. In: IFIP International Workshop on Information Security Theory and Practice. pp. 40–56. Springer (2014). https://doi.org/10.1007/978-3-662-43826-8_4
9. Bronchain, O., Cassiers, G.: 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)
10. Carlet, C., Daif, A., Guilley, S., Tavernier, C.: Quasi-linear masking to protect against both sca and fia. Cryptology ePrint Archive (2023)
11. Chari, S., Jutla, C.S., Rao, J.R., Rohatgi, P.: 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)
12. Chen, L., Chen, L., Jordan, S., Liu, Y.K., Moody, D., Peralta, R., Perlner, R.A., Smith-Tone, D.: Report on post-quantum cryptography, vol. 12. US Department of Commerce, National Institute of Standards and Technology ... (2016)
13. Dworkin, M.J.: Sha-3 standard: Permutation-based hash and extendable-output functions. NIST FIPS (2015). https://doi.org/10.6028/NIST.FIPS.202
14. Fritzmann, T., Van Beirendonck, M., Roy, D.B., Karl, P., Schamberger, T., Verbauwhede, I., Sigl, G.: Masked accelerators and instruction set extensions for post-quantum cryptography. Cryptology ePrint Archive (2021). https://doi.org/10.46586/tches.v2022.i1.414-460
15. Fujisaki, E., Okamoto, T.: Secure integration of asymmetric and symmetric encryption schemes. In: Annual international cryptology conference. pp. 537–554. Springer (1999). https://doi.org/10.1007/3-540-48405-1_34
16. Goubin, L.: A sound method for switching between boolean and arithmetic masking. In: Cryptographic Hardware and Embedded Systems—CHES 2001: Third International Workshop Paris, France, May 14–16, 2001 Proceedings 3. pp. 3–15. Springer (2001). https://doi.org/10.1007/3-540-44709-1_2

17. Goudarzi, D., Joux, A., Rivain, M.: How to securely compute with noisy leakage in quasilinear complexity. In: International Conference on the Theory and Application of Cryptology and Information Security. pp. 547–574. Springer (2018). https://doi.org/10.1007/978-3-030-03329-3_19
18. Goudarzi, D., Prest, T., Rivain, M., Vergnaud, D.: Probing security through input-output separation and revisited quasilinear masking. IACR Transactions on Cryptographic Hardware and Embedded Systems pp. 599–640 (2021). https://doi.org/10.46586/tches.v2021.i3.599-640
19. Heinz, D., Kannwischer, M.J., Land, G., Pöppelmann, T., Schwabe, P., Sprenkels, D.: First-order masked kyber on arm cortex-m4. Cryptology ePrint Archive (2022)
20. Heinz, D., Pöppelmann, T.: Combined fault and dpa protection for lattice-based cryptography. IEEE Transactions on Computers **72**(4), 1055–1066 (2022). https://doi.org/10.1109/TC.2022.3197073
21. Karatsuba, A.: Multiplication of multidigit numbers on automata. In: Soviet physics doklady. vol. 7, pp. 595–596 (1963)
22. Kocher, P.C.: 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). https://doi.org/10.1007/3-540-68697-5_9
23. Mosca, M., Piani, M.: 2021 quantum threat timeline report global risk institute. Global Risk Institute (2022)
24. Ngo, K.: Side-Channel Analysis of Post-Quantum Cryptographic Algorithms. Ph.D. thesis, KTH Royal Institute of Technology (2023)
25. Oder, T., Schneider, T., Pöppelmann, T., Güneysu, T.: Practical cca2-secure and masked ring-lwe implementation. Cryptology ePrint Archive (2016). https://doi.org/10.13154/tches.v2018.i1.142-174
26. Rains, E.M., Sloane, N.J.: Self-dual codes. arXiv preprint math/0208001 (2002). https://doi.org/10.48550/arXiv.math/0208001
27. Ravi, P., Roy, S.S.: Side-channel analysis of lattice-based pqc candidates. In: Round 3 Seminars, NIST Post Quantum Cryptography (2021)
28. Saarinen, M.J.: Intro to side-channel security of nist pqc standards. In: PQC Seminars, NIST (2023)
29. Shor, P.W.: Algorithms for quantum computation: discrete logarithms and factoring. In: Proceedings 35th annual symposium on foundations of computer science. pp. 124–134. Ieee (1994). https://doi.org/10.1109/SFCS.1994.365700
30. Wang, W., Méaux, P., Cassiers, G., Standaert, F.X.: Efficient and private computations with code-based masking. IACR Transactions on Cryptographic Hardware and Embedded Systems pp. 128–171 (2020). https://doi.org/10.13154/tches.v2020.i2.128-171
31. Wang, Y., Zhu, X.: A fast algorithm for the fourier transform over finite fields and its vlsi implementation. IEEE Journal on Selected Areas in Communications **6**(3), 572–577 (1988). https://doi.org/10.1109/49.1926