Higher-Order Masked Saber

Suparna Kundu(✉), Jan-Pieter D'Anvers, Michiel Van Beirendonck, Angshuman Karmakar, and Ingrid Verbauwhede

imec-COSIC, KU Leuven, Kasteelpark Arenberg 10, Bus 2452, 3001 Leuven-Heverlee, Belgium
{suparna.kundu,jan-pieter.danvers,michiel.beirendonck, angshuman.karmakar,ingrid.verbauwhede}@esat.kuleuven.be

Abstract. Side-channel attacks are formidable threats to the cryptosystems deployed in the real world. An effective and provably secure countermeasure against side-channel attacks is masking. In this work, we present a detailed study of higher-order masking techniques for the key-encapsulation mechanism Saber. Saber is one of the lattice-based finalist candidates in the National Institute of Standards of Technology's post-quantum standardization procedure. We provide a detailed analysis of different masking algorithms proposed for Saber in the recent past and propose an optimized implementation of higher-order masked Saber. Our proposed techniques for first-, second-, and third-order masked Saber have performance overheads of 2.7x, 5x, and 7.7x respectively compared to the unmasked Saber. We show that compared to Kyber which is another lattice-based finalist scheme, Saber's performance degrades less with an increase in the order of masking. We also show that higher-order masked Saber needs fewer random bytes than higher-order masked Kyber. Additionally, we adapt our masked implementation to uSaber, a variant of Saber that was specifically designed to allow an efficient masked implementation. We present the first masked implementation of uSaber, showing that it indeed outperforms masked Saber by at least 12% for any order. We provide optimized implementations of all our proposed masking schemes on ARM Cortex-M4 microcontrollers.

Keywords: Post-quantum cryptography · Higher-order masking · Saber · Key-encapsulation mechanism

1 Introduction

The security of public-key cryptography (PKC) is dependent on the computational intractability of some underlying mathematical problems. The current most widely used public-key cryptographic algorithms RSA [44] and elliptic curve cryptography (ECC) [37] are based on the hardness of large integer factorization problem and elliptic curve discrete logarithm problem respectively. Unfortunately, both of these hard problems can be solved in polynomial time with large-scale quantum computers by using Shor's [46] and Proos-Zalka's [41] algorithm. Post-quantum cryptography (PQC) is a branch of PKC that focuses

C. Galdi and S. Jarecki (Eds.): SCN 2022, LNCS 13409, pp. 93–116, 2022.
https://doi.org/10.1007/978-3-031-14791-3_5

on designing cryptographic algorithms whose underlying mathematical problems remain hard even in the presence of large quantum computers. Considering the fast evolution of quantum computers and their impending threat to our current public-key infrastructure, the National Institute of Standards and Technology (NIST) started a procedure to standardize post-quantum public-key cryptographic primitives such as digital signatures, public-key encryption, and key-encapsulation mechanism in 2016 [39].

In 2020, NIST announced four finalists and five alternative candidates for the post-quantum key-encapsulation mechanism (KEM) category, that advanced to the 3rd round [2]. Three of the four finalist KEMs: Saber [19], Kyber [10], and NTRU [26], are lattice-based. NTRU is an NTRU-based KEM, whereas Kyber and Saber are based on variants of the learning with errors (LWE) problem. The security of Kyber can be reduced to module learning with errors (MLWE) problem, and the security of Saber is based on module learning with rounding (MLWR) problem. The hardness of both LWE and LWR problems are dependent on the difficulty to solve a set of noisy linear equations. This noise is explicitly added for a LWE problem but is implicitly generated in a LWR problem using the round-off of a few least significant bits.

Initially, the main focus of the NIST post-quantum standardization procedure was the mathematical security of the schemes, together with the performance, and the memory footprint of the cryptographic implementation in embedded devices. With the advancement of the standardization process, the focus was broadened to take into account the implementation-security of the schemes also. Side-channel attacks (SCA) [34] are a well-known type of physical attacks against implementations of cryptographic algorithms. These attacks exploit leakage of information, such as timing information, power consumption, electromagnetic radiation, etc., which leaks information from the physical device which runs the algorithm to extract the secret key.

Silverman et al. [47] first showed a timing attack on quantum secure lattice-based cryptographic protocol NTRUEncrypt [28] by exploiting the non-constant time implementation. To prevent the timing attack, most of the cryptographic protocols use constant-time implementation, including Saber and Kyber. In recent years, many works [3,24,29,42,50] showed SCA on lattice-based cryptographic schemes with the help of power consumption and electromagnetic leakage information. A provably-secure countermeasure against these kinds of SCA is masking [13].

The masking technique can also provide security against higher-order attacks, where the adversary can use the power consumption information of multiple points. However, the performance cost of the masked scheme increases with the order of SCA. Reparaz et al. [43] were the first to introduce a first-order SCA resistant masked implementation of chosen-plaintext attack (CPA) secure ring-LWE based decryption. Nevertheless, real-world applications use chosen-ciphertext attack (CCA) secure cryptosystems. Lattice-based quantum secure KEMs such as Saber and Kyber achieve CCA security by using a variant of Fujisaki-Okamoto transformation [30] on their CPA secure design. Oder et al. [40] proposed a 1st-order CCA secure masked Ring-LWE key decapsulation and

reported an overhead factor of 5.2x in performance over an unmasked implementation on an ARM Cortex-M4.

Van Beirendonck et al. [6] proposed the first-order SCA secure implementation of Saber with an overhead factor of 2.5x. This performance was achievable because of the power-of-two moduli and efficient utilization of masking techniques specifically aimed at first-order security [48]. Heinz et al. [25] presented an optimized first-order masked implementation of Kyber with an overhead factor of 3.4x compared to the unmasked implementation of Kyber. Fritzmann et al. [21] proposed first-order masked implementations of Kyber and Saber with instruction set extensions, and Bos et al. [11] proposed higher-order masked implementations of Kyber.

First-order masked implementations of schemes are typically vulnerable against higher-order side-channel attacks [36,49], i.e., the attacks that exploit side-channel leakages of multiple intermediate values. Ngo et al. [38] proposed an attack on the first-order masked Saber using a deep neural network constructed at the profiling stage. This attack does not violate the assumption of the first-order masked Saber but exploits higher-order side-channel leakages. Higher-order masking increases the noise level exponentially and prevents attacks that exploit higher-order side-channel leakages.

In the third-round of the NIST submission, the Saber team introduced uSaber as a variant of Saber. In uSaber, the secrets are sampled from a uniform distribution instead of a centered binomial distribution as used in Saber. The authors claim that the advantage of this modification is twofold. First, it makes the scheme simpler since sampling from a uniform distribution is more straightforward than sampling from a centered binomial distribution, and it also reduces the modulus by a factor of two. Second, this change allows a very efficient masking of the secret values. However, this claim is yet to be proven as there exists no masked implementation of uSaber to corroborate this claim.

Contribution. In this work, we provide arbitrary-order masked implementations of Saber and uSaber, and we compare their performances with the state-of-the-art masked implementations of Saber and Kyber. We are the first to propose a higher-order masked implementation of uSaber. For this, we present a masked centered uniform sampler which is then applied to uSaber instead of Saber's centered binomial sampler. We generally take advantage of Saber's power-of-two moduli to mask both Saber's and uSaber's decapsulation algorithm, and we compare different recently proposed algorithms for ciphertext comparison in higher-order masked settings.

We implement and benchmark our higher-order masked Saber and uSaber on an ARM Cortex-M4 microcontroller using the PQM4 framework. The first-, second-, and third-order masked decapsulation algorithm of Saber has an overhead factor of 2.7x, 5x, and 7.7x over the unmasked implementation, respectively. In uSaber, the overhead factor for first-order is 2.3x, second-order is 4.2x, and third-order is 6.5x compared to the unmasked version. We include the performance results and requisite of the random bytes during masking for each masked primitive of first-, second-, and third-order masked Saber and uSaber.

Our implementations are available at https://github.com/KULeuven-COSIC/Higher-order-masked-Saber.

Finally, we compare the performances of our higher-order masked implementations of Saber and uSaber with the higher-order masked implementations of Kyber and Saber presented in [11,12]. We demonstrate that the performances of masked Saber implementations outperform masked Kyber implementations. Further, we show that the performance of masked uSaber is better and requires fewer random bytes than masked Saber and Kyber for any order.

2 Preliminaries

2.1 Notation

We denote the ring of integers modulo q by $\mathbb{Z}_q$ and the quotient ring $\mathbb{Z}_q[X]/(X^{256}+1)$ by R_q. We use R_q^l to represent the ring which contains vectors with l elements of R_q. The ring with $l \times l$ matrices over R_q is denoted by $R_q^{l\times l}$. We use lower case letters to denote single polynomials, bold lower case letters to denote vectors and bold upper case letters to denote matrices. The j-th coefficient of the polynomial c is represented as $c[j]$, where $j \in \{0, 1, \ldots, 255\}$. The j-th coefficient of the i-th polynomial of the vector $\mathbf{b}$ is represented as $\mathbf{b}[i][j]$, where $j \in \{0, 1, \ldots, 255\}$ and $i \in \{0, 1, \ldots, l-1\}$. Sometimes the set of $(n+1)$ elements $\{x_0, x_1, \ldots, x_n\}$ from the same ring R is denoted by $\{x_i\}_{0\leq i\leq n}$.

The rounding operation is denoted by $\lfloor \cdot \rceil$, and it returns the closest integer with ties rounded upwards. The operations $x \ll b$ and $x \gg b$ denote the logical shifting of x by b positions left and right, respectively. These operations are extended on polynomials by performing them coefficientwise.

We denote $x \leftarrow \chi(S)$ when x is sampled from the set S according to the distribution χ. We use the notation $x \leftarrow \chi(S, seed_x)$ to represent that x belongs to the set S and is generated by the pseudorandom number generator χ with the help of seed $seed_x$. To represent the uniform distribution we use $\mathcal{U}$. The centered binomial distribution is denoted by β_μ with standard deviation $\sqrt{\mu/4}$. The centered uniform distribution is expressed as $\mathcal{U}_u$, when it samples uniformly from $[-2^{(u-1)}, 2^{(u-1)}-1]$. We use $\mathtt{HW}(x)$ to represent the Hamming weight of x.

2.2 Saber

In this section, we introduce the Saber encryption scheme. The parameter set of Saber includes three power-of-two moduli q, p and t, which define the rings R_q, R_p and R_t used in the algorithm. From these moduli, one can calculate the number of bits of one coefficient as $\epsilon_q = \log_2(q)$, $\epsilon_p = \log_2(p)$ and $\epsilon_t = \log_2(t)$. The parameter set also includes a vector length l, which increases with increase in security, and an integer μ defining the coins of the secret distribution β_μ. Given a set of parameters, the key generation, encryption, and decryption of Saber are shown in Fig. 1. For an in-depth review of the Saber encryption scheme, we refer to the original paper [19,20].

2.3 uSaber

uSaber or uniform-Saber was proposed in third round NIST submission [20] as a variant of Saber. The principal alteration in uSaber from Saber is that it uses a centered uniform distribution $\mathcal{U}_u$ for sampling secret vectors instead of the centered binomial distribution β_μ. The coefficients in polynomials of secret vector are from $[-2^{(u-1)}, 2^{(u-1)}-1]$ rather than $[-\mu/2, \mu/2]$. Due to this modification, uSaber receives approximately the same level of security as Saber with a slightly reduced parameters set as shown in Table 1.

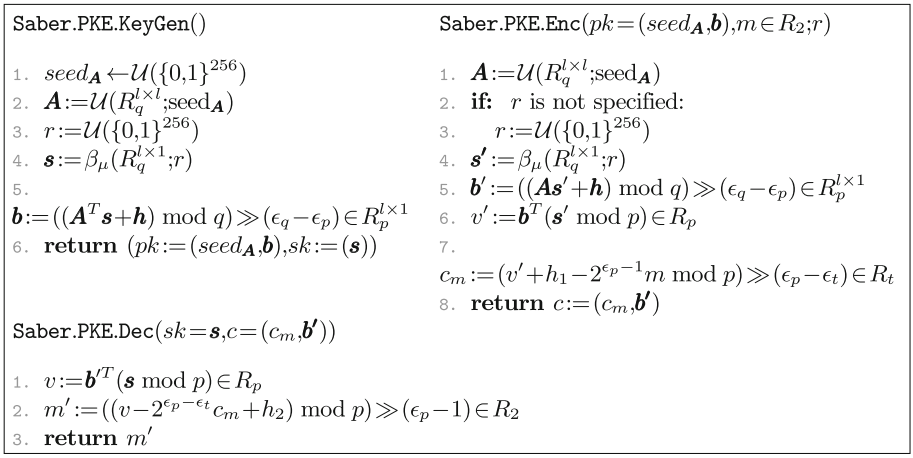

Fig. 1. Saber.PKE

Table 1. Parameters of Saber and uSaber with security and failure probability

Scheme	Parameters			Post-quantum Security	Failure Probability	NIST Security Level
	Identical	Different				
		q	Secret Distribution			
uSaber	$l = 3$, $p = 2^{10}$ $n = 256$, $t = 2^4$	$\mathbf{2^{12}}$	$\mathcal{U}_\mathbf{2}$	2^{165}	2^{-167}	3
Saber		$\mathbf{2^{13}}$	$\boldsymbol{\beta_8}$	2^{172}	2^{-136}	3

2.4 Fujisaki-Okamoto Transformation

The encryption scheme outlined in the previous section only provides security against passive attackers (IND-CPA security). One can obtain active security (IND-CCA) security by using a generic transformation such as a post-quantum version of the Fujisaki-Okamoto transformation [22,27]. The idea is that the

encapsulation encrypts a random input, and also uses this input as a seed for all randomness. The decapsulation can then decrypt the seed from the ciphertext and recompute the ciphertext. This recomputed ciphertext can then be used to check if the input ciphertext is generated correctly. The Fujisaki-Okamoto transformation transforms the encryption scheme into a key encapsulation mechanism (KEM). Given hash functions $\mathcal{F}$, $\mathcal{G}$ and $\mathcal{H}$, the saber KEM is given in Fig. 2. Again, we refer to the original Saber paper [19,20] for a more detailed description.

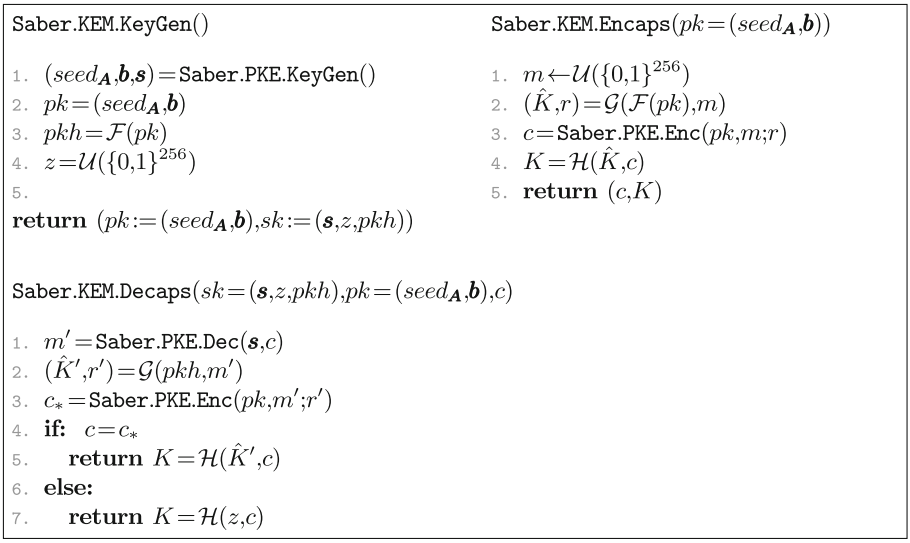

Fig. 2. Saber.KEM

2.5 Higher-Order Masking

Masking is a widely used countermeasure against side-channel attacks. The nth-order masked scheme can provide security against at most nth-order differential power attacks. The general idea of nth-order masking is to split the sensitive variable x into $n+1$ shares and then perform all the operations of the algorithms on each of the shares individually. The shares of the sensitive variable look uniformly random and the sensitive information can only be retrieved after combining all the $n+1$ shares. Moreover, if an adversary can get side-channel information from at most n points, he will not learn anything about the sensitive variable. In an nth-order masked implementation, linear operations typically duplicate $(n+1)$ times, and non-linear operations need to use more complex and costlier methods. As a consequence, the performance cost of a nth-order masked implementation increases at least by a factor of $(n+1)$.

There are several methods for masking. We primarily deal with two kinds of masking techniques: arithmetic masking and Boolean masking. For both the

masking techniques, in order to obtain nth-order security, the sensitive variable $x \in \mathbb{Z}_q$ needs to be split into $n+1$ independent shares $x_0, x_1, \ldots, x_n \in \mathbb{Z}_q$. In arithmetic masking, the relation between the sensitive variable x and the $n+1$ shares of x is $x = x_0 + x_1 + \cdots + x_n \bmod q$. Whereas, in Boolean masking the sensitive variable x and its $n+1$ shares are related as $x = x_0 \oplus x_1 \oplus \cdots \oplus x_n$.

The arithmetic masking is advantageous for protecting arithmetic operations such as addition, subtraction, multiplication. For example, to protect the modular addition $z = x + y \bmod q$ against n-order attacks, when only x contains sensitive data, we split x into $n+1$ shares $\{x_i\}_{0 \le i \le n}$ such that $\sum_{i=0}^{n} x_i \bmod q = x$, then the shares of $z = \sum_{i=0}^{n} z_i \bmod q$ are:

$$z_i = \begin{cases} x_i + y \bmod q, & \text{if } i = 0 \\ x_i, & \text{if } 1 \le i \le n \end{cases}.$$

If x and y both contains sensitive data, we split y together with x into $n+1$ shares $\{y_i\}_{0 \le i \le n}$ such that $\sum_{i=0}^{n} y_i = y \bmod q$, then the shares of $z = \sum_{i=0}^{n} z_i \bmod q$ are:

$$z_i = x_i + y_i \bmod q, \ 0 \le i \le n.$$

To securely compute the multiplication $z = x \cdot y \bmod q$, when x only contains sensitive data, we create $n+1$ shares $\{x_i\}_{0 \le i \le n}$ for x such that $\sum_{i=0}^{n} x_i \bmod q = x$, then the shares of $z = \sum_{i=0}^{n} z_i \bmod q$ are:

$$z_i = x_i \cdot y \bmod q, \ 0 \le i \le n.$$

We prefer Boolean masking for variables that undergo bitwise operations. For example, if we want to perform logical shift operation $z = x \gg l$ securely, write x into $n+1$ shares $\{x_i\}_{0 \le i \le n}$ such that $\oplus_{i=0}^{n} x_i = x$, then calculate $z_i = x_i \gg l, \forall i$ to obtain the shares of $z = \oplus_{i=0}^{n} z_i$.

3 Masking Saber

In a key encapsulation mechanism (KEM), the secret key remains fixed for a significant amount of time. Specifically, the decapsulation algorithm uses the non-ephemeral secret key $\mathbf{s}$, and therefore it is the most susceptible operation against side-channel attacks. In this paper, we focus on protecting the non-ephemeral secret key of Saber during the decapsulation. We introduce a masked decapsulation algorithm for Saber, which can resist higher-order side-channel attacks. The decapsulation procedure of Saber can be partitioned into three segments, namely decryption, re-encryption, and ciphertext comparison. For visualization, we present the flow of Saber's decapsulation algorithm in Fig. 3. Here, all the modules that process sensitive data due to the involvement with the secret have been marked grey. These modules are vulnerable from the perspective of side-channel attacks and need to be masked. In this section, we describe all the masked primitives that are used in the higher-order masked decapsulation procedure of Saber. We also present a new algorithm to perform the ciphertext comparison component in the masked setting. We will go through each part of the decapsulation algorithm of Saber chronologically and explain the methods we have used to mask them.

3.1 Arithmetic Operations

The decapsulation algorithm of Saber is heavily dependent on polynomial arithmetic, such as polynomial addition/subtraction and polynomial multiplication. We use arithmetic masking to protect these operations. As shown in Fig. 3, the decapsulation algorithm requires the following operations: addition between one masked and another unmasked polynomial, addition between two masked polynomials, and multiplication between one masked and another unmasked polynomial. For masking these operations, we follow the methods described in Sect. 2.5.

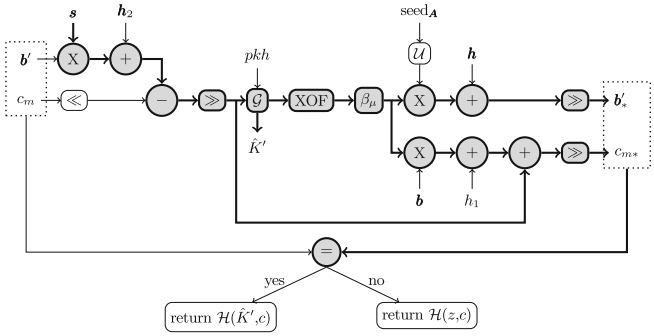

Fig. 3. Decapsulation of Saber. In grey the operations that are influenced by the long term secret $\boldsymbol{s}$ and thus vulnerable to side-channel attacks [6].

To perform the polynomial multiplication, the original unmasked Saber multiplication uses a hybrid multiplication, a combination of Toom-Cook-4, 2 levels of Karatsuba, and school-book multiplication [19,33,35]. We use this same multiplication technique in our masked implementation. Chung et al. [14] have recently introduced an efficient method to perform polynomial multiplication by using the number-theoretic transform. The same method could be used for the implementation of masked Saber to provide a significant performance improvement [12]. However, this is not the goal of our work and we keep this as future work.

3.2 Compression

In the last step of `Saber.PKE.Dec`, m is computed by calculating the most significant bit (`MSB`) for each coefficient. It compresses each coefficient of the polynomial $(v - 2^{\epsilon_p - \epsilon_t} c_m + h_2) \bmod p$ to produce a polynomial m where each coefficient is one bit long.

The logical shift operation is easy on Boolean shares. In this situation, we need to apply a logical shift operation on each share separately. Unfortunately, computing this logical shift operation on arithmetic shares is not trivial. This fact is discussed elaborately in [6] for the case of first-order masking, and the similar issue arises for higher-order masking also.

We compute `MSB` on arithmetic shares by taking the following steps: first, convert arithmetic shares to Boolean shares (A2B conversion), second, perform logical shift operation on Boolean shares, and finally, return to arithmetic domain

with the Boolean to arithmetic (B2A) conversion. As $m \in R_2$, the resultant polynomial after compression is a polynomial with 1 bit coefficients. Here, the Boolean shares of m act like arithmetic shares of m. Therefore, we do not need the B2A conversion step.

Bitslicing is a technique that helps to improve the performance of bitwise operations. We have opted for the algorithm proposed in [15] using the bitsliced implementation of [17] for the A2B conversion of our implementations.

3.3 Masked Hashing

In Saber, the hash function $\mathcal{G}$ and the pseudo-random number generator `XOF` are realized by using `SHA3-512` and `SHAKE-128`, respectively. Both are different instances of the sponge function `Keccak-f[1600]` [7]. It has been shown that this construction is easy to protect by using Boolean masking [23].

`Keccak-f[1600]` permutation has five steps: θ, ρ, π, χ and ι. In between these θ, ρ, π are linear diffusion steps and ι is a simple addition. As all of these four are linear operations on Boolean shares, we just need to apply them for each share. χ is a degree 2 non-linear mapping and therefore requires extra attention to apply masking. Gross et al. [23] developed a technique to implement χ in the higher-order mask setting. We have adopted their technique in our implementations.

3.4 Masked Centered Binomial Sampler

`Saber.PKE.Enc` uses the centered binomial sampler for sampling the vector $\mathbf{s}'$. This sampler outputs the result of $\texttt{HW}(x) - \texttt{HW}(y)$, where x and y are pseudo-random numbers of bit length four. These pseudorandom numbers are generated by using `SHAKE-128`. As mentioned in Sect. 3.3, `SHAKE-128` is protected by using Boolean masking. After the generation of $\mathbf{s}'$, polynomial multiplications with $\mathbf{s}'$ (e.g. $\mathbf{A}\mathbf{s}'$ and $\mathbf{b}^T\mathbf{s}'$) take place. `SHAKE-128` creates Boolean shares, but polynomial multiplication is an arithmetic operation that is less expensive with arithmetic shares. To mitigate this issue, we need to include a conversion algorithm that converts Boolean shares into arithmetic shares (B2A conversion) in the masked centered binomial sampler.

Schneider et al. [45] propose two efficient higher-order masked centered binomial samplers: $sampler_1$ computes masked shares bitwise, whereas $sampler_2$ uses the bitslicing techniques to improve throughput. We have adopted the implementation of $sampler_2$ together with the modification made by Van Beirendonck et al. [6] specifically for Saber.

To convert shares from Boolean to arithmetic, we use the B2A algorithm proposed in [8]. The details have been provided in the Algorithm 1. In this Algorithm, `SecBitAdd` calculates shares of $\texttt{HW}(x)$ and `SecBitSub` takes shares of $\texttt{HW}(x)$ and shares of y as inputs and outputs shares of $z = \texttt{HW}(x) - \texttt{HW}(y)$. The function `SecConstAdd` adds $\mu/2 = 4$ with the shares of z to avoid any negative value that can occur after `SecBitSub`. In the next step the `B2A` function converts all the Boolean shares of z to the arithmetic shares of A and the last step converts shares of A from $\{0, 1, \ldots, 8\}$ to $\{-4, -3, \ldots, 3, 4\}$.

Algorithm 1: Masked centered binomial sampler [45]

Input : $\{x_i\}_{0\le i\le n}, \{y_i\}_{0\le i\le n}$ where $x_i, y_i \in \mathbb{R}_2^\kappa$ such that $\bigoplus_{i=0}^{n} x_i = x, \bigoplus_{i=0}^{n} y_i = y$

Output : $\{A_i\}_{0\le i\le n}$ where $A_i \in \mathbb{R}_q$ and $\sum_{i=0}^{n} A_i = (\texttt{HW}(x) - \texttt{HW}(y)) \bmod q$

1 $\{z_i\}_{0\le i\le n} \leftarrow \texttt{SecBitAdd}(\{x_i\}_{0\le i\le n})$ [6]
2 $\{z_i\}_{0\le i\le n} \leftarrow \texttt{SecBitSub}(\{z_i\}_{0\le i\le n}, \{y_i\}_{0\le i\le n})$ [45]
3 $\{z_i\}_{0\le i\le n} \leftarrow \texttt{SecConstAdd}(\{z_i\}_{0\le i\le n})$ [6]
4 $\{A_i\}_{0\le i\le n} \leftarrow \texttt{B2A}(\{z_i\}_{0\le i\le n})$ [8]
5 $A_1 \leftarrow (A_1 - \mu/2) \bmod q$
6 **return** $\{y_i\}_{0\le i\le n}$

3.5 Masked Comparison

The masked ciphertext comparison component is required to check the equality between masked ciphertext generated from re-encryption and the public ciphertext. This step performs the equality check $c \stackrel{?}{=} c^*$ of the `Saber.KEM.Decaps` algorithm in the masked domain.

An easy but efficient method for the first-order masked comparison is introduced by Oder et al. [40]. Unfortunately, this hash-based method is limited to first-order masking, and cannot be generalized to check ciphertext equality in the higher-order masked settings.

Different approaches for higher-order masked comparison were recently analyzed thoroughly by D'Anvers et al. [17]. In general, there are four approaches. The *simple method* originally due to Barthe et al. [5] groups individual bits into a large `SecOR` operation. This requires a pre-processing step to handle ciphertext compression that is straightforward to mask for Saber, but more complex for Kyber [21]. The *arithmetic method* was developed in a series of works [4,9,18], and aims to reduce the total number of comparisons by grouping coefficients into a random sum. The *decompression method* [11] developed for Kyber avoids masking the compression of the re-encrypted ciphertext, by decompressing the input ciphertext instead. Finally, the *hybrid method* [16] introduced the idea of using different of the previously discussed methods for the different components of the ciphertext. All of these approaches rely on A2B conversions, which can be heavily optimized using bitslicing [12,17].

In this section, we will discuss two of these different approaches to higher-order masked comparison. The first one is the Saber-adapted decompression method, which was not considered in [17]. The second one is the simple method, which was found to be the most efficient method for Saber in that same work. For both methods, we consolidate concurrent A2B optimization techniques proposed in [12,17].

3.5.1 Decompressed Masked Comparison Algorithm

Bos et al. [11] introduced a new method based on A2B conversion for the masked comparison algorithm for Kyber, in order to reduce the cost of the Boolean equality check circuit. This method does not perform the compression operation on

Algorithm 2: Decompressed masked comparison algorithm

Input : $\{\mathbf{b}'_i\}_{0\le i\le n}$ where each $\mathbf{b}'_i \in \mathbb{R}^l_{2^{\epsilon_q}}$ and $\bigoplus_{i=0}^{n} \mathbf{b}'_i = \mathbf{b}'$,
$\{c'_i\}_{0\le i\le n}$ where each $c'_i \in \mathbb{R}_{2^{\epsilon_p}}$ and $\bigoplus_{i=0}^{n} c'_i = c'$,
publicly available $\mathbf{b}$ and c_m

Output : $\{bit_i\}_{0\le i\le n}$ with each $bit_i \in \{0,1\}$ such that $\bigoplus_{i=0}^{n} bit_i = 1$ iff $\mathbf{b} = \mathbf{b}' \gg (\epsilon_q - \epsilon_p)$ and $c_m = c' \gg (\epsilon_p - \epsilon_t)$, else 0

1 //For $\mathbf{b}$ part of ciphertext
2 $\mathbf{s_b} \leftarrow (\mathbf{b} \ll (\epsilon_q - \epsilon_p)) - 1$ //Decompression operation on $\mathbf{b}$
3 $\mathbf{e_b} \leftarrow (\mathbf{b} \ll (\epsilon_q - \epsilon_p)) + 2^{(\epsilon_q - \epsilon_p)}$
4 $\{\mathbf{b}''_i\}_{0\le i\le n} \leftarrow \{\mathbf{b}'_i\}_{0\le i\le n}$
5 $\mathbf{b}''_1 \leftarrow \mathbf{b}'_1 - \mathbf{s_b} + 2^{(\epsilon_q - 1)}$
6 $\mathbf{b}'_1 \leftarrow \mathbf{b}'_1 - \mathbf{e_b}$
7 $\{\mathbf{y}'_i\}_{0\le i\le n} \leftarrow \texttt{A2B}(\{\mathbf{b}''_i\}_{0\le i\le n})$
8 $\{\mathbf{y}_i\}_{0\le i\le n} \leftarrow \texttt{A2B}(\{\mathbf{b}'_i\}_{0\le i\le n})$
9 $\{\mathbf{y}_i\}_{0\le i\le n} \leftarrow \texttt{MSB}(\{\mathbf{y}_i\}_{0\le i\le n}) || \texttt{MSB}(\{\mathbf{y}'_i\}_{0\le i\le n})$
10 //For c_m part of ciphertext
11 $s_{c_m} \leftarrow (c_m \ll (\epsilon_p - \epsilon_t)) - 1$//Decompression operation on c_m
12 $e_{c_m} \leftarrow (c_m \ll (\epsilon_p - \epsilon_t)) + 2^{(\epsilon_p - \epsilon_t)}$
13 $\{c''_i\}_{0\le i\le n} \leftarrow \{c'_i\}_{0\le i\le n}$
14 $c''_1 \leftarrow c'_1 - s_{c_m} + 2^{(\epsilon_p - 1)}$
15 $c'_1 \leftarrow c'_1 - e_{c_m}$
16 $\{x'_i\}_{0\le i\le n} \leftarrow \texttt{A2B}(\{c''_i\}_{0\le i\le n})$
17 $\{x_i\}_{0\le i\le n} \leftarrow \texttt{A2B}(\{c'_i\}_{0\le i\le n})$
18 $\{x_i\}_{0\le i\le n} \leftarrow \texttt{MSB}(\{x_i\}_{0\le i\le n}) || \texttt{MSB}(\{x'_i\}_{0\le i\le n})$
19 //Boolean circuit to test all bits of each coefficient of $(\mathbf{y}, x)$ is 1
20 $\{bit_i\}_{0\le i\le n} \leftarrow \texttt{BooleanAllBitsOneTest}\ (\{\mathbf{y}_i\}_{0\le i\le n}, \{x_i\}_{0\le i\le n}, 2, 2)$
21 **return** $\{bit_i\}_{0\le i\le n}$

the recomputed ciphertext, but performs a decompression operation on the public ciphertext instead. Then, the comparison is performed in the uncompressed domain. The decompressed operation is less costly to apply on public ciphertext, as it is public and so this operation can be performed unmasked.

Let us assume the public ciphertext be $c = (\mathbf{b}, c_m)$, where $\mathbf{b}$ be the key contained part and c_m be the message contained part of the ciphertext c. In Saber, the compression operation is applied to generate the ciphertext during encryption, and this operation is a many-to-one operation. In this process, each coefficient of b loses three bits, and each coefficient of c_m loses six bits. So, as compensation for the masked comparison, we use a decompression operation, which outputs an interval of integers for each coefficient of the public ciphertext. Let, $c[j]$ be a coefficient of the public ciphertext c, and the corresponding output of decompression operation be $(s_c[j], e_c[j])$. This implies every element in between the interval $(s_c[j], e_c[j])$ becomes $c[j]$ after the compression operation.

Next, we verify that each coefficient of the shared uncompressed ciphertext of c^* which is generated from the re-encryption, lies in the corresponding decom-

Algorithm 3: BooleanAllBitsOneTest

Input : $\{\mathbf{y}_i\}_{0\le i\le n}$ where each $\mathbf{y}_i \in \mathbb{R}^l_{2^{\texttt{bmod1}}}$ and $\bigoplus_{i=0}^{n} \mathbf{y}_i = \mathbf{y}$,
$\{x_i\}_{0\le i\le n}$ where each $x_i \in \mathbb{R}_{2^{\texttt{bmod2}}}$ and $\bigoplus_{i=0}^{n} x_i = x$,
bmod1, and bmod2

Output : $\{bit_i\}_{0\le i\le n}$ with each $bit_i \in \{0,1\}$ such that $\bigoplus_{i=0}^{n} bit_i = 1$ iff each bit of every coefficients of $\mathbf{y}$ and x is 1, else 0

1 **for** $\mathtt{j}_1 = 1$ *to* l **do**
2 **for** $s = 1$ *to* bmod1 **do**
3 $\{\mathbf{u}_i[s][j_1]\}_{0\le i\le n} \leftarrow$ Bitslice$(\{\mathbf{y}_i^{(s)}[j_1]\}_{0\le i\le n})$
4 **for** $s = 1$ *to* bmod2 **do**
5 $\{v_i[s]\}_{0\le i\le n} \leftarrow$ Bitslice$(\{x_i^{(s)}\}_{0\le i\le n})$
6 //Secure And on both
7 $\{w_i\}_{0\le i\le n} \leftarrow \{v_i[1]\}_{0\le i\le n}$
8 **for** $s = 2$ *to* bmod2 **do**
9 $\{w_i\}_{0\le i\le n} \leftarrow$ SecAnd$(\{w_i\}_{0\le i\le n}, \{v_i[s]\}_{0\le i\le n})$
10 **for** $j = 1$ *to* l **do**
11 $\{\mathbf{y}_i[j]\}_{0\le i\le n} \leftarrow \{\mathbf{u}_i[1][j]\}_{0\le i\le n}$
12 **for** $s = 2$ *to* bmod1 **do**
13 $\{\mathbf{y}_i[j]\}_{0\le i\le n} \leftarrow$ SecAnd$(\{\mathbf{y}_i[j]\}_{0\le i\le n}, \{\mathbf{u}_i[s][j]\}_{0\le i\le n})$
14 $\{w_i\}_{0\le i\le n} \leftarrow$ SecAnd$(\{w_i\}_{0\le i\le n}, \mathbf{y}_i[j]\}_{0\le i\le n})$
15 **for** $j = \log_2(256) - 1$ *to* 0 **do**
16 $\{w'_i\}_{0\le i\le n} \leftarrow w_{0\le i\le n} \gg 2^j$
17 $\{w_i\}_{0\le i\le n} \leftarrow w_{0\le i\le n} \bmod (2^{2^j})$
18 $\{w_i\}_{0\le i\le n} \leftarrow$ SecAnd$(\{w_i\}_{0\le i\le n}, \{w'_i\}_{0\le i\le n})$
19 $\{bit_i\}_{0\le i\le n} \leftarrow \{w_i\}_{0\le i\le n}$
20 **return** $\{bit_i\}_{0\le i\le n}$

pressed interval. Let $c^*[j]$ be the arithmetically masked uncompressed ciphertext coefficient corresponding to the public ciphertext coefficient $c[j]$. Now, if $c^*[j] \in (s_c[j], e_c[j])$ for all coefficients j, then the comparison returns success and outputs the shared valid key else returns a random invalid key. The test $c^*[j] \stackrel{?}{\in} (s_c[j], e_c[j])$ is realized by checking whether $c^*[j] - s_c[j]$ is a positive number and whether $c^*[j] - e_c[j]$ is a negative number. We have adopted this method for performing the higher-order masked ciphertext comparison in Saber as shown in Algorithm 2.

In Algorithm 2, lines 2–3 and 11–12 compute the start-point and the end-point of the interval for each coefficient of the key contained part $\mathbf{b}$ and the message contained part c_m of the public ciphertext c, respectively. The MSB of any number acts as a sign bit, i.e., if the MSB is 1 then the number is negative, else the number is positive. As in an ideal case, $c^*[j] - s_c[j] > 0$ and $c^*[j] - e_c[j] < 0$, so the MSB$(c^*[j] - s_c[j])$ should be 0 and the MSB$(c^*[j] - e_c[j])$ should be 1. In order to avoid two different kinds of checking for $c^*[j] - s_c[j]$ and $c^*[j] - e_c[j]$,

we need to add a constant l with $c^*[j] - s_c[j]$ such that its MSB becomes 1. The value of l is $2^{(\epsilon_q - 1)}$ and $2^{(\epsilon_p - 1)}$ for $\mathbf{b}$ and c_m part of c, respectively. We compute the MSB of an arithmetically masked variable in the following way: we convert the arithmetic shares to Boolean shares using A2B conversion, and then we use a shift operation to extract the masked shares of MSB. Finally, Algorithm 2 uses Algorithm 3, the BooleanAllBitsOneTest function to combine the output bits of all coefficients and returns a single-bit indicating success or failure.

3.5.2 Simple Masked Comparison Algorithm

Next, we describe the simple method as given in [17]. Note that the re-encrypted ciphertext c^* is arithmetically masked and uncompressed, but the public ciphertext c is unmasked and compressed. As mentioned earlier, our task is to verify whether c equals c^* after compression. In this method, we perform the following steps: firstly, we transform arithmetic shares of c^* to Boolean shares by using A2B conversion algorithm, secondly, we compress the re-encrypted Boolean masked ciphertext c^* by using coefficientwise logical right shift, thirdly, we subtract the public ciphertext c from the compressed and masked re-encrypted ciphertext c^*. This method is shown in Algorithm 4.

Algorithm 4: Simple masked comparison algorithm [16]

Input : $\{\mathbf{b}'_i\}_{0 \le i \le n}$ where each $\mathbf{b}'_i \in \mathbb{R}^l_{2^{\epsilon_q}}$ and $\bigoplus_{i=0}^{n} \mathbf{b}'_i = \mathbf{b}'$,
$\{c'_i\}_{0 \le i \le n}$ where each $c'_i \in \mathbb{R}_{2^{\epsilon_p}}$ and $\bigoplus_{i=0}^{n} c'_i = c'$,
publicly available $\mathbf{b}$ and c_m

Output : $\{bit_i\}_{0 \le i \le n}$ with each $bit_i \in \{0, 1\}$ such that $\bigoplus_{i=0}^{n} bit_i = 1$ iff $\mathbf{b} = \mathbf{b}' \gg (\epsilon_q - \epsilon_p)$ and $c_m = c' \gg (\epsilon_p - \epsilon_t)$, else 0

1 //For $\mathbf{b}$ part of ciphertext
2 $\{\mathbf{y}_i\}_{0 \le i \le n} \leftarrow$ A2B$(\{\mathbf{b}'_i\}_{0 \le i \le n})$
3 $\{\mathbf{y}_i\}_{0 \le i \le n} \leftarrow (\{\mathbf{y}_i\}_{0 \le i \le n} \gg (\epsilon_q - \epsilon_p))$
4 $\mathbf{y}_1 \leftarrow \mathbf{y}_1 \oplus \mathbf{b}$
5 //For c_m part of ciphertext
6 $\{x_i\}_{0 \le i \le n} \leftarrow$ A2B$(\{c'_i\}_{0 \le i \le n})$
7 $\{x_i\}_{0 \le i \le n} \leftarrow (\{x_i\}_{0 \le i \le n} \gg (\epsilon_p - \epsilon_t))$
8 $x_1 \leftarrow x_1 \oplus c_m$
9 //Boolean circuit to test all bits of each coefficient of $(\mathbf{y}, x)$ is 0
10 $\mathbf{y}_1 \leftarrow \neg \mathbf{y}_1$
11 $x_1 \leftarrow \neg x_1$
12 $\{bit_i\}_{0 \le i \le n} \leftarrow$ BooleanAllBitsOneTest $(\{\mathbf{y}_i\}_{0 \le i \le n}, \{x_i\}_{0 \le i \le n}, \epsilon_p, \epsilon_t)$
13 **return** $\{bit_i\}_{0 \le i \le n}$

3.5.3 Bitsliced A2B

Both the decompression method and the simple method rely heavily on A2B conversions. Throughout the implementations, we use the bitsliced A2B conver-

sion [17] for further speed-up. Moreover, `A2B` conversions use the `SecAdd` subfunction to perform masked addition. Bronchain et al. [12] proposed a `SecAdd` which uses $k-1$ `SecAnd` operations for k-bit inputs, as opposed to $2k-3$ `SecAnd` operations required in [17]. We included this technique into the implementation of [17] to receive better performance.

4 Masking uSaber

In uSaber, the coefficients of the secret vector are sampled according to the centered uniform distribution $\mathcal{U}_u$ instead of the centered binomial distribution β_μ. Here, the hamming weight computation of the centered binomial distribution is replaced by the sign extension of u bits to ϵ_q bits, to generate a sample in $[-2^{(u-1)}, 2^{(u-1)}-1]$ from u uniformly random bits. This secret vector sampler is the only component that differs between Saber and uSaber. A main advantage of uSaber is that the centered uniform sampler has fewer operations compared to the centered binomial sampler and therefore, is easier to mask.

Similar to the centered binomial sampler, the centered uniform sampler takes pseudorandom Boolean-masked bits as input that are produced by the masked `SHAKE-128` function. Our simple higher-order masked centered uniform sampler is shown in Algorithm 5. We base it on `SecConsAdd` in the masked centered binomial sampler, mentioned in Sect. 3.4. First, we use `xor` to transform a negative number into a positive number. Second, we apply `B2A` conversion algorithm to convert Boolean shares to arithmetic shares. Third, we subtract 2^{u-1} from the arithmetic shares to map them from $[0, 2^u-1]$ back to $[-2^{(u-1)}, 2^{(u-1)}-1]$. This masked sampler does not require `SecBitAdd` and `SecBitSub` which are used in masked centered binomial sampler. The centered uniform sampler is simpler and requires fewer masked operations than the centered binomial sampler.

Algorithm 5: Masked centered uniform Sampler

Input : $\{x_i\}_{0\le i\le n}$ where $x_i \in \mathbb{R}_2^u$ such that $\bigoplus_{i=0}^{n} x_i = x$
Output : $\{A_i\}_{0\le i\le n}$ where $A_i \in \mathbb{R}_q$ and $\sum_{i=0}^{n} A_i = (x \oplus 2^{u-1}) - 2^{u-1} \mod q$

1 $\{z_0\} \leftarrow (\{z_0\} \oplus 2^{u-1})$
2 $\{A_i\}_{0\le i\le n} \leftarrow \text{B2A}(\{z_i\}_{0\le i\le n})$ [8]
3 $A_1 \leftarrow A_1 - 2^{u-1} \mod q$
4 **return** $\{y_i\}_{0\le i\le n}$

5 Performance Evaluation

To demonstrate the performance of all of the proposed methods, we implement them on a 32-bit ARM Cortex-M4 microcontroller, STM32F407-DISCOVERY development board. We adopt the widely used PQM4 [32] post-quantum cryptographic library and benchmarking framework for performance evaluation. In this

framework, the system timer (SisTick) is used to measure the cycle counts. This framework uses a 24 MHz main system clock and a 48 MHz TRNG clock. We take advantage of the on-chip TRNG for sampling masking randomness instead of generating in advance and storing random bits in a table like Kyber [11]. This TRNG generates 32 random bits per 40 TRNG clock cycles, which is equal to 20 main system clock cycles. We include the cost of randomness sampling with the benchmarks. We use `arm-none-eabi-gcc` version `9.2.1` to accomplish the measurements of our implementations.

5.1 Performance Analysis of Comparison Algorithms for Saber

We present the cycle counts of the implementation for arbitrary order masked comparison algorithms of Saber. In Saber, the parameters are: $\epsilon_q = 13$, $\epsilon_p = 10$, $\epsilon_t = 4$ and $l = 3$. We break down the cycle counts into three parts: spent during all the `A2B` conversion, spent in computing the function `BooleanAllBitsOneTest` for the corresponding parameter, and spent in performing all other operations. Table 2 contains the performance details of masked ciphertext algorithms presented in Sects. 3.5.1 and Sects. 3.5.2.

In Table 2, we include two versions of the decompressed comparison algorithm and the simple comparison algorithm. We use the bitsliced `A2B` conversion technique of masked simple comparator proposed in [17] for the first version, and we improve this bitsliced `A2B` converter by employing the technique introduced in [12]. It can be seen from the table the performance of the decompressed comparison algorithm gains 9%, 16%, and 17% improvements for first-, second-, and third-order masking after using [12], respectively. Side by side, the improved decompressed comparison algorithm requires 21% fewer random bytes for any order masking. The performance of the simple comparison algorithm improves by 8%, 15%, and 16% for first-, second-, and third-order masking after using [12], respectively. The improved simple comparison algorithm requires approximately 19% fewer random bytes for any order masking.

As we can see from the table, the cycle count for all `A2B` conversions employed in the decompressed comparison algorithm is almost double for all orders compared to the simple comparison algorithm. However, for all the orders, the clock cycle required to compute the function `BooleanAllBitsOneTest` with corresponding parameters is approximately one-fourth in the decompressed comparison algorithm than the simple comparison algorithm. As we can see from Table 2, the improved simple comparison algorithm is approximately 43% faster and employ roughly 42% fewer random bytes than the improved decompressed comparison algorithm for any order masking of Saber. Similar results can be found for the higher-order masked uSaber. So, we use the improved simple comparison algorithm in our higher-order masked Saber and uSaber decapsulation algorithms.

5.2 Performance Analysis for Masked Saber Decapsulation

We present the performance cost of the Saber algorithm for higher-order masking in Table 3. This table also provides the breakdown of the performance cost of the higher-order masking for all the modules of the masked Saber decapsulation

algorithm. As mentioned earlier, for masked Saber implementations we use the hybrid polynomial multiplication, a combination of Toom-Cook-4, Karatsuba, and schoolbooks multiplication. Therefore, we use the Saber implementation which uses the hybrid polynomial multiplication to get the overhead factor for n-th order masked Saber. To maintain simplicity, most of the implementation is written in C. Only the hybrid multiplication is in assembly and generated by using the optimal implementation proposed in [31].

From Table 3, we can see that the performance overhead factor of masked Saber decapsulation implementation for first-order is 2.69x, for second-order is 4.96x, and for third-order is 7.71x. From the table, we can see that the overhead factor for arithmetic operations approximately is $(n + 1)$ for nth-order masking due to $n+1$ time repetitions of each operation. On the other hand, the non-linear operations on arithmetic shares, for example, hash functions, binomial sampler, compression operation, and ciphertext comparison, have much larger overhead factors in the masked setting. To maintain the security assumption, we need to use random bytes in some masking algorithms (example: SecAnd, SecAdd, SecRefresh, etc.). Table 4 shows random bytes requirements for all the components of the

Table 2. Performance breakdown of the implementation of masked comparison algorithms in the Cortex-M4 platform.

Masked Comparator	CPU [k]cycles		
	1st	2nd	3rd
Decompressed comparison [This work]	651	2,107	3,606
all `A2B` conversion	612	2,047	3,518
`BooleanAllBitsOneTest`	9	29	50
Other operations	28	31	37
# random bytes	12,048	47,920	95,840
Improved decompressed comparison [This work]	**588**	**1,756**	**2,962**
all `A2B` conversion	549	1,696	2,875
`BooleanAllBitsOneTest`	9	29	50
Other operations	28	31	37
# random bytes	**9,424**	**37,424**	**74,848**
Simple comparison [17]	363	1,160	1,992
all `A2B` conversion	308	1,023	1,766
`BooleanAllBitsOneTest`	38	117	202
Other operations	16	19	24
# random bytes	6,992	26,864	53,728
Improved simple comparison [This work]	**331**	**985**	**1,671**
all `A2B` conversion	276	848	1,444
`BooleanAllBitsOneTest`	38	117	202
Other operations	16	19	24
# random bytes	**5,680**	**21,616**	**43,232**

Table 3. Performance cost of all the modules of the higher-order masked decapsulation procedure of Saber.

Order	CPU [k]cycles					
	No mask	1st		2nd		3rd	
Saber-Decapsulation	1,121	3,022	(2.69x)	5,567	(4.96x)	8,649	(7.71x)
CPA-PKE-Decryption	129	297	(2.30x)	527	(4.08x)	775	(6.00x)
Polynomial arithmetic	126	237	(1.88x)	349	(2.76x)	464	(3.68x)
Compression	2	59	(29.50x)	178	(89.00x)	310	(155.00x)
Hash $\mathcal{G}$ (SHA3-512)	13	123	(9.46x)	242	(18.61x)	379	(29.15x)
CPA-PKE-Encryption	853	2,477	(2.90x)	4,670	(5.47x)	7,370	(8.64x)
Secret generation	69	909	(13.17x)	1,995	(28.91x)	3,561	(51.60x)
XOF (SHAKE-128)	63	611	(9.69x)	1,210	(19.20x)	1,887	(29.95x)
CBD (Binomial Sampler)	6	297	(49.50x)	785	(130.83x)	1,674	(279.00x)
Polynomial arithmetic	783	1,235	(2.00x)	1,688	(3.41x)	2,136	(4.86x)
Polynomial Comparison		331		985		1,671	
Other operations	126	126	(1.00x)	126	(1.00x)	126	(1.00x)

Table 4. Randomness cost of all the modules of the higher-order masked decapsulation algorithm of Saber.

Order	# Random bytes		
	1st	2nd	3rd
Saber-Decapsulation	12,752	43,760	93,664
CPA-PKE-Decryption	928	3,712	7,424
Polynomial arithmetic	0	0	0
Compression	928	3,712	7,424
Hash $\mathcal{G}$ (SHA3-512)	192	576	1,152
CPA-PKE-Encryption	11,312	38,512	83,168
Secret generation	5,952	17,856	41,856
XOF (SHAKE-128)	960	2880	5,760
CBD (Binomial Sampler)	4,992	14,976	36,096
Polynomial arithmetic	0	0	0
Polynomial Comparison	5,680	21,616	43,232
Other operations	0	0	0

higher-order masked Saber decapsulation algorithm. It can be seen from Table 4 that the random bytes requirement increases with the order. The first-order implementation requires 12k random bytes, the second-order and third-order implementations require 43k (3.43x) and 93k (7.34x) random bytes, respectively.

5.3 Performance Analysis for Masked uSaber Decapsulation

The performance cost and breakdown of the performance cost of the higher-order masking for all the modules of the masked uSaber decapsulation algorithm are

presented in Table 5. As we mentioned before, the main advantage of uSaber against Saber is the coefficients of the secret vector are sampled from $\mathcal{U}_2$ instead of β_8. Thanks to the parameter choice of secret distribution in uSaber, it needs fewer numbers of pseudorandom bits than Saber. This fact reduces the cycle cost of `XOF` by almost 60% for the unmasked version of uSaber compared to Saber. Another advantage is that the hamming weight computation of μ bits in the centered binomial sampler β_μ is swapped by the sign extension of u bits in the centered uniform sampler $\mathcal{U}_u$. It reduces the performance cost of the secret sampler in unmasked uSaber by 50% compared to Saber. Altogether, the secret generation is almost 59% faster for the unmasked decapsulation algorithm of uSaber compared to Saber. The performance cost of the secret generation is lower in uSaber compared to Saber also after integrating masking. The performances of the secret generation in masked uSaber are 55%, 52%, and 45% faster compared to masked Saber for first-, second-, and third-order, respectively. Additionally, the value of q for uSaber is 2^{12}, whereas it is 2^{13} for Saber. This factor reduces one bit in the `A2B` conversion for uSaber during the masked polynomial comparison. It makes the masked polynomial comparison 5%, 5%, and 2% faster in uSaber than Saber for first-, second-, and third-order, respectively.

As we can observe from Table 5, the approximate performance overhead factor of masked uSaber decapsulation implementation for first-order is 2.32x, for second-order is 4.19x, and for third-order is 6.54x. Table 6 presents random bytes requirements for all the segments of the higher-order masked uSaber decapsulation. We obtain from Table 6 that here also the random bytes requirement grows with the order of masking. The first-order implementation needs 10k random bytes, the second-order and third-order implementations use 36k (3.49x) and 79k (7.57x) random bytes, respectively.

Table 5. Performance cost of all the modules of higher-order masked decapsulation procedure of uSaber.

Order	CPU [k]cycles						
	No mask	1st		2nd		3rd	
uSaber-Decapsulation	1,062	2,473	(2.32x)	4,452	(4.19x)	6,947	(6.54x)
CPA-PKE-Decryption	130	297	(2.28x)	527	(4.05x)	775	(5.96x)
Polynomial arithmetic	128	237	(1.85x)	349	(2.72x)	464	(3.62x)
Compression	2	59	(29.50x)	178	(89.00x)	310	(155.00x)
Hash $\mathcal{G}$ (SHA3-512)	13	122	(9.38x)	242	(18.61x)	379	(29.15x)
CPA-PKE-Encryption	791	1,928	(2.43x)	3,556	(4.49x)	5,667	(7.16x)
Secret generation	28	400	(14.28x)	954	(34.07x)	1,928	(68.85x)
XOF (SHAKE-128)	25	245	(9.80x)	484	(19.36x)	756	(30.24x)
Uniform distribution	3	155	(51.66x)	469	(156.33x)	1,172	(390.66x)
Polynomial arithmetic	763	1,214	(2.00x)	1,667	(3.40x)	2,114	(4.89x)
Polynomial Comparison		313		934		1,623	
Other operations	126	126	(1.00x)	126	(1.00x)	126	(1.00x)

Table 6. Randomness cost of all the modules of higher-order masked decapsulation algorithm of uSaber.

Order	# Random bytes		
	1st	2nd	3rd
uSaber-Decapsulation	10,544	36,848	79,840
CPA-PKE-Decryption	928	3,712	7,424
Polynomial arithmetic	0	0	0
Compression	928	3,712	7,424
Hash $\mathcal{G}$ (SHA3-512)	192	576	1,152
CPA-PKE-Encryption	9,104	31,600	69,344
Secret generation	4,032	12,096	30,336
XOF (SHAKE-128)	960	2880	5,760
Uniform distribution	3,072	9,216	24,576
Polynomial arithmetic	0	0	0
Polynomial Comparison	5,392	20,464	40,928
Other operations	0	0	0

5.4 Comparison with State-of-the-Art

In this section, we compare our masked Saber and uSaber implementations with the state-of-the-art masked implementations of Saber and Kyber. We present the performances of our masked implementations in the Cortex-M4 platform and present them in Table 7. Bronchain et al. [12] introduced faster implementations of higher-order masked `A2B` and `B2A` conversion utilizing bitsliced techniques and used these conversions to propose higher-order masking implementations of Saber and Kyber. The performances of Bronchain et al.'s masked Saber and Kyber implementations in the Cortex-M4 platform are presented in Table 7. As we mentioned before, the integration of NTT multiplication in masked Saber can provide a significant performance boost. In [12], authors use NTT multiplication for Saber to receive better performance. In order to use NTT multiplication, the authors use a multi-moduli approach that extends the modulus [1]. Even so, the performance of our 1st, 2nd and 3rd order masked implementations of Saber achieve 39%, 23%, and 13% improvement than their masked implementation of Saber, respectively.

In [11], Bos et al. proposed higher-order masked implementations of Kyber. The masked kyber implementation in [11] is faster and uses fewer random bytes than the implementation of masked kyber presented in [12] only for first-order because this masked Kyber uses an optimized implementation for first-order, while Bronchain et al.'s one uses the generalized one. The performance for 2nd and 3rd order masked implementations of Kyber in [12] receives 73% and 85% improvement over the masked Kyber of [11], respectively. However, our implementation of masked Saber is faster than masked Kyber presented in [12] 60% for first-order, 53% for second-order, and 48% for third-order. Also, the performance

of our first-order masked Saber is 3% faster than the optimized implementation of the first-order masked Kyber presented in [11]. In terms of random bytes requirement, our masked Saber receives factor 20.61x and 25.98x improvement over the masked Kyber in [11] for 2nd and 3rd order masked implementations, respectively.

As discussed in Sect. 4, masked uSaber uses less number of operations and random numbers than masked Saber due to the choice of secret distribution and parameters in uSaber. Table 7 shows the performances of higher-order masked implementations of uSaber. Further, this table contains the performance of first-order masked Saber [6] and first-order masked Kyber [25], which are specially optimized to prevent the first-order differential power attacks. We can observe from Table 7 that our generalized implementation of first-order masked uSaber is 12% faster than the optimized implementation of masked Saber and is 16% faster than the optimized implementation of masked Kyber. The implementation of masked uSaber is faster than the fastest implementation of higher-order masked Saber 20% for second-order and 19% for third-order. Masked uSaber also needs 15% less random numbers for second-order and 14% less random numbers for third-order over masked Saber. In conclusion, we observe from the reported results of Table 7 that higher-order masked uSaber achieves better performance and needs fewer random bytes than masked Saber and Kyber for any order.

Table 7. The comparison between Saber and Kyber regarding the performance and the random bytes requirement.

Scheme		Performance CPU [k]cycles				/ Random bytes		
		Unmask	1st	2nd	3rd	1st	2nd	3rd
uSaber	This paper	1,062	**2,473**	**4,452**	**6,947**	**10,544**	**36,848**	**79,840**
Saber	This paper	1,121	**3,022**	**5,567**	**8,649**	**12,752**	**43,760**	**93,664**
	[12]	773	5,027	7,320	9,988	-	-	-
	[6] †	1,123	2,833	-	-	11,656	-	-
Kyber	[11] †	882	3,116*	44,347	115,481	12,072*	902,126	2,434,170
	[12]	804	7,716	11,880	16,715	-	-	-
	[25] †	816	2,978	-	-	-	-	-

†: measurements are taken from the paper
*: uses optimized implementation for first-order masking

6 Conclusions

Saber is often touted as very helpful for masking because of its two unique design components, the power-of-two moduli, and the MLWR problem. Van Beirendonck et al. [6] showed the first-order masked Saber receives better performance and needs fewer random bytes than the first-order masked Kyber. In our work, we substantiated this claim for arbitrary higher-order masking and show that the higher-order masked Saber also acquires better performance and requires fewer random bytes for its design decisions.

The third round submission document of Saber claims that the design decisions behind uSaber will be further beneficial for masking even compared to Saber. This work first concretely justifies those design decisions.

Furthermore, integrating our methods of masking is not dependent upon the underlying polynomial multiplication, which is one of the computationally expensive components. Our masked implementations can be adapted for Saber or uSaber that use the NTT multiplication instead of the hybrid multiplication.

Acknowledgements. This work was supported in part by CyberSecurity Research Flanders with reference number VR20192203, the Research Council KU Leuven (C16/15/058), the Horizon 2020 ERC Advanced Grant (101020005 Belfort) and SRC grant 2909.001.

Jan-Pieter D'Anvers and Angshuman Karmakar are funded by FWO (Research Foundation - Flanders) as junior post-doctoral fellows (contract numbers 133185/1238822N LV and 203056/1241722N LV). Michiel Van Beirendonck is funded by FWO as Strategic Basic (SB) PhD fellow (project number 1SD5621N).

References

1. Abdulrahman, A., Chen, J., Chen, Y., Hwang, V., Kannwischer, M.J., Yang, B.: Multi-moduli NTTs for saber on Cortex-M3 and Cortex-M4. IACR Trans. Cryptogr. Hardw. Embed. Syst. **2022**(1), 127–151 (2022). https://doi.org/10.46586/tches.v2022.i1.127-151
2. Alagic, G., et al.: Status Report on the Second Round of the NIST Post-Quantum Cryptography Standardization Process (2020). https://nvlpubs.nist.gov/nistpubs/ir/2020/NIST.IR.8309.pdf
3. Amiet, D., Curiger, A., Leuenberger, L., Zbinden, P.: Defeating NewHope with a single trace. Cryptology ePrint Archive, Report 2020/368 (2020). https://ia.cr/2020/368
4. Bache, F., Paglialonga, C., Oder, T., Schneider, T., Güneysu, T.: High-speed masking for polynomial comparison in lattice-based KEMs. IACR Trans. Cryptogr. Hardw. Embed. Syst. **2020**(3), 483–507 (2020). https://doi.org/10.13154/tches.v2020.i3.483-507
5. Barthe, G., et al.: Masking the GLP lattice-based signature scheme at any order. In: Nielsen, J.B., Rijmen, V. (eds.) EUROCRYPT 2018, Part II. LNCS, vol. 10821, pp. 354–384. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-78375-8_12
6. Beirendonck, M.V., D'Anvers, J.P., Karmakar, A., Balasch, J., Verbauwhede, I.: A side-channel resistant implementation of SABER. Cryptology ePrint Archive, Report 2020/733 (2020). https://ia.cr/2020/733
7. Bertoni, G., Daemen, J., Peeters, M., Van Assche, G.: Keccak. In: Johansson, T., Nguyen, P.Q. (eds.) EUROCRYPT 2013. LNCS, vol. 7881, pp. 313–314. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-38348-9_19
8. Bettale, L., Coron, J., Zeitoun, R.: Improved high-order conversion from Boolean to arithmetic masking. IACR Trans. Cryptogr. Hardw. Embed. Syst. **2018**(2), 22–45 (2018). https://doi.org/10.13154/tches.v2018.i2.22-45
9. Bhasin, S., D'Anvers, J., Heinz, D., Pöppelmann, T., Beirendonck, M.V.: Attacking and defending masked polynomial comparison for lattice-based cryptography. IACR Trans. Cryptogr. Hardw. Embed. Syst. **2021**(3), 334–359 (2021). https://doi.org/10.46586/tches.v2021.i3.334-359

10. Bos, J., et al.: CRYSTALS - Kyber: a CCA-secure module-lattice-based KEM. Cryptology ePrint Archive, Report 2017/634 (2017). https://ia.cr/2017/634
11. Bos, J.W., Gourjon, M., Renes, J., Schneider, T., van Vredendaal, C.: Masking Kyber: first- and higher-order implementations. IACR Cryptology ePrint Archive, p. 483 (2021). https://eprint.iacr.org/2021/483
12. Bronchain, O., Cassiers, G.: Bitslicing arithmetic/Boolean masking conversions for fun and profit with application to lattice-based KEMs. Cryptology ePrint Archive, Report 2022/158 (2022). https://ia.cr/2022/158
13. Chari, S., Jutla, C.S., Rao, J.R., Rohatgi, P.: Towards sound approaches to counteract power-analysis attacks. In: Wiener, M. (ed.) CRYPTO 1999. LNCS, vol. 1666, pp. 398–412. Springer, Heidelberg (1999). https://doi.org/10.1007/3-540-48405-1_26
14. Chung, C.M., Hwang, V., Kannwischer, M.J., Seiler, G., Shih, C., Yang, B.: NTT multiplication for NTT-unfriendly rings new speed records for saber and NTRU on Cortex-M4 and AVX2. IACR Trans. Cryptogr. Hardw. Embed. Syst. **2021**(2), 159–188 (2021). https://doi.org/10.46586/tches.v2021.i2.159-188
15. Coron, J.-S., Großschädl, J., Vadnala, P.K.: Secure conversion between Boolean and arithmetic masking of any order. In: Batina, L., Robshaw, M. (eds.) CHES 2014. LNCS, vol. 8731, pp. 188–205. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-662-44709-3_11
16. Coron, J.S., Gérard, F., Montoya, S., Zeitoun, R.: High-order polynomial comparison and masking lattice-based encryption. Cryptology ePrint Archive, Report 2021/1615 (2021). https://ia.cr/2021/1615
17. D'Anvers, J.P., Beirendonck, M.V., Verbauwhede, I.: Revisiting higher-order masked comparison for lattice-based cryptography: algorithms and bit-sliced implementations. Cryptology ePrint Archive, Report 2022/110 (2022). https://ia.cr/2022/110
18. D'Anvers, J.P., Heinz, D., Pessl, P., van Beirendonck, M., Verbauwhede, I.: Higher-order masked ciphertext comparison for lattice-based cryptography. Cryptology ePrint Archive, Report 2021/1422 (2021). https://ia.cr/2021/1422
19. D'Anvers, J.-P., Karmakar, A., Sinha Roy, S., Vercauteren, F.: Saber: module-LWR based key exchange, CPA-secure encryption and CCA-secure KEM. In: Joux, A., Nitaj, A., Rachidi, T. (eds.) AFRICACRYPT 2018. LNCS, vol. 10831, pp. 282–305. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-89339-6_16
20. D'Anvers, J.P., et al.: SABER. Technical report, National Institute of Standards and Technology (2020). https://csrc.nist.gov/projects/post-quantum-cryptography/round-3-submissions
21. Fritzmann, T., et al.: Masked accelerators and instruction set extensions for post-quantum cryptography. IACR Trans. Cryptogr. Hardw. Embed. Syst. **2022**(1), 414–460 (2021). https://doi.org/10.46586/tches.v2022.i1.414-460. https://tches.iacr.org/index.php/TCHES/article/view/9303
22. Fujisaki, E., Okamoto, T.: Secure integration of asymmetric and symmetric encryption schemes. In: Wiener, M. (ed.) CRYPTO 1999. LNCS, vol. 1666, pp. 537–554. Springer, Heidelberg (1999). https://doi.org/10.1007/3-540-48405-1_34
23. Gross, H., Schaffenrath, D., Mangard, S.: Higher-order side-channel protected implementations of Keccak. Cryptology ePrint Archive, Report 2017/395 (2017). https://ia.cr/2017/395
24. Guo, Q., Johansson, T., Nilsson, A.: A key-recovery timing attack on post-quantum primitives using the Fujisaki-Okamoto transformation and its application on FrodoKEM. Cryptology ePrint Archive, Report 2020/743 (2020). https://ia.cr/2020/743

25. Heinz, D., Kannwischer, M.J., Land, G., Pöppelmann, T., Schwabe, P., Sprenkels, D.: First-order masked Kyber on ARM Cortex-M4. IACR Cryptology ePrint Archive, p. 58 (2022). https://eprint.iacr.org/2022/058
26. Hoffstein, J., Pipher, J., Silverman, J.H.: NTRU: a ring-based public key cryptosystem. In: Buhler, J.P. (ed.) ANTS 1998. LNCS, vol. 1423, pp. 267–288. Springer, Heidelberg (1998). https://doi.org/10.1007/BFb0054868
27. Hofheinz, D., Hövelmanns, K., Kiltz, E.: A modular analysis of the Fujisaki-Okamoto transformation. In: Kalai, Y., Reyzin, L. (eds.) TCC 2017. LNCS, vol. 10677, pp. 341–371. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-70500-2_12
28. Howgrave-Graham, N., Silverman, J.H., Whyte, W.: Choosing parameter sets for NTRUEncrypt with NAEP and SVES-3. Cryptology ePrint Archive, Report 2005/045 (2005). https://ia.cr/2005/045
29. Huang, W.L., Chen, J.P., Yang, B.Y.: Power analysis on NTRU prime. Cryptology ePrint Archive, Report 2019/100 (2019). https://ia.cr/2019/100
30. Jiang, H., Zhang, Z., Chen, L., Wang, H., Ma, Z.: IND-CCA-secure key encapsulation mechanism in the quantum random oracle model, revisited. Cryptology ePrint Archive, Report 2017/1096 (2017). https://ia.cr/2017/1096
31. Kannwischer, M.J., Rijneveld, J., Schwabe, P.: Faster multiplication in $\mathbb{Z}_{2^m}[x]$ on Cortex-M4 to speed up NIST PQC candidates. Cryptology ePrint Archive, Report 2018/1018 (2018). https://ia.cr/2018/1018
32. Kannwischer, M.J., Rijneveld, J., Schwabe, P., Stoffelen, K.: PQM4: post-quantum crypto library for the ARM Cortex-M4. https://github.com/mupq/pqm4
33. Karmakar, A., Mera, J.M.B., Roy, S.S., Verbauwhede, I.: Saber on ARM CCA-secure module lattice-based key encapsulation on ARM. IACR Trans. Cryptogr. Hardw. Embed. Syst. **2018**(3), 243–266 (2018). https://doi.org/10.13154/tches.v2018.i3.243-266
34. Kocher, P.C.: Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In: Koblitz, N. (ed.) CRYPTO 1996. LNCS, vol. 1109, pp. 104–113. Springer, Heidelberg (1996). https://doi.org/10.1007/3-540-68697-5_9
35. Mera, J.M.B., Karmakar, A., Verbauwhede, I.: Time-memory trade-off in Toom-Cook multiplication: an application to module-lattice based cryptography. IACR Trans. Cryptogr. Hardw. Embed. Syst. **2020**(2), 222–244 (2020). https://doi.org/10.13154/tches.v2020.i2.222-244
36. Messerges, T.S.: Using second-order power analysis to attack DPA resistant software. In: Koç, Ç.K., Paar, C. (eds.) CHES 2000. LNCS, vol. 1965, pp. 238–251. Springer, Heidelberg (2000). https://doi.org/10.1007/3-540-44499-8_19
37. Miller, V.S.: Use of elliptic curves in cryptography. In: Williams, H.C. (ed.) CRYPTO 1985. LNCS, vol. 218, pp. 417–426. Springer, Heidelberg (1986). https://doi.org/10.1007/3-540-39799-X_31
38. Ngo, K., Dubrova, E., Guo, Q., Johansson, T.: A side-channel attack on a masked IND-CCA secure saber KEM implementation. IACR Trans. Cryptogr. Hardw. Embed. Syst. **2021**(4), 676–707 (2021). https://doi.org/10.46586/tches.v2021.i4.676-707
39. NIST: Post-Quantum Cryptography Standardization. https://csrc.nist.gov/Projects/Post-Quantum-Cryptography/Post-Quantum-Cryptography-Standardization
40. Oder, T., Schneider, T., Pöppelmann, T., Güneysu, T.: Practical CCA2-secure and masked ring-LWE implementation. IACR Trans. Cryptogr. Hardw. Embed. Syst. **2018**(1), 142–174 (2018). https://doi.org/10.13154/tches.v2018.i1.142-174

41. Proos, J., Zalka, C.: Shor's discrete logarithm quantum algorithm for elliptic curves. Quantum Inf. Comput. **3**(4), 317–344 (2003). https://doi.org/10.26421/QIC3.4-3
42. Ravi, P., Bhasin, S., Roy, S.S., Chattopadhyay, A.: Drop by Drop you break the rock - exploiting generic vulnerabilities in Lattice-based PKE/KEMs using EM-based Physical Attacks. Cryptology ePrint Archive, Report 2020/549 (2020). https://ia.cr/2020/549
43. Reparaz, O., Sinha Roy, S., Vercauteren, F., Verbauwhede, I.: A masked ring-LWE implementation. In: Güneysu, T., Handschuh, H. (eds.) CHES 2015. LNCS, vol. 9293, pp. 683–702. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-48324-4_34
44. Rivest, R.L., Shamir, A., Adleman, L.M.: A method for obtaining digital signatures and public-key cryptosystems. Commun. ACM **21**(2), 120–126 (1978). http://doi.acm.org/10.1145/359340.359342
45. Schneider, T., Paglialonga, C., Oder, T., Güneysu, T.: Efficiently masking binomial sampling at arbitrary orders for lattice-based crypto. In: Lin, D., Sako, K. (eds.) PKC 2019. LNCS, vol. 11443, pp. 534–564. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-17259-6_18
46. Shor, P.W.: Algorithms for quantum computation: discrete logarithms and factoring. In: 35th Annual Symposium on Foundations of Computer Science, Santa Fe, New Mexico, USA, 20–22 November 1994, pp. 124–134. IEEE Computer Society (1994). https://doi.org/10.1109/SFCS.1994.365700
47. Silverman, J.H., Whyte, W.: Timing attacks on NTRUEncrypt via variation in the number of hash calls. In: Abe, M. (ed.) CT-RSA 2007. LNCS, vol. 4377, pp. 208–224. Springer, Heidelberg (2006). https://doi.org/10.1007/11967668_14
48. Van Beirendonck, M., D'Anvers, J.P., Verbauwhede, I.: Analysis and comparison of table-based arithmetic to Boolean masking. **2021**(3), 275–297 (2021). https://doi.org/10.46586/tches.v2021.i3.275-297. https://tches.iacr.org/index.php/TCHES/article/view/8975
49. Waddle, J., Wagner, D.: Towards efficient second-order power analysis. In: Joye, M., Quisquater, J.-J. (eds.) CHES 2004. LNCS, vol. 3156, pp. 1–15. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-28632-5_1
50. Xu, Z., Pemberton, O., Roy, S.S., Oswald, D., Yao, W., Zheng, Z.: Magnifying side-channel leakage of lattice-based cryptosystems with chosen ciphertexts: the case study of Kyber. Cryptology ePrint Archive, Report 2020/912 (2020). https://ia.cr/2020/912