On the Masking-Friendly Designs for Post-quantum Cryptography

Suparna Kundu[1(✉)], Angshuman Karmakar[1,2], and Ingrid Verbauwhede[1]

[1] COSIC, KU Leuven, Kasteelpark Arenberg 10, Bus 2452, 3001 Leuven-Heverlee, Belgium
{suparna.kundu,angshuman.karmakar,ingrid.verbauwhede}@esat.kuleuven.be
[2] Indian Institute of Technology Kanpur, Kanpur, India

Abstract. Masking is a well-known and provably secure countermeasure against side-channel attacks. However, due to additional redundant computations, integrating masking schemes is expensive in terms of performance. The performance overhead of integrating masking countermeasures is heavily influenced by the design choices of a cryptographic algorithm and is often not considered during the design phase.

In this work, we deliberate on the effect of design choices on integrating masking techniques into lattice-based cryptography. We select Scabbard, a suite of three lattice-based post-quantum key-encapsulation mechanisms (KEM), namely Florete, Espada, and Sable. We provide arbitrary-order masked implementations of all the constituent KEMs of the Scabbard suite by exploiting their specific design elements. We show that the masked implementations of Florete, Espada, and Sable outperform the masked implementations of Kyber in terms of speed for any order masking. Masked Florete exhibits a 73%, 71%, and 70% performance improvement over masked Kyber corresponding to the first-, second-, and third-order. Similarly, Espada exhibits 56%, 59%, and 60% and Sable exhibits 75%, 74%, and 73% enhanced performance for first-, second-, and third-order masking compared to Kyber respectively. Our results show that the design decisions have a significant impact on the efficiency of integrating masking countermeasures into lattice-based cryptography.

Keywords: Post-quantum cryptography · Key-encapsulation mechanism · Side-channel attacks · Scabbard · Higher-order masking

1 Introduction

Physical attacks such as fault injection and side-channel attacks are potent threats to any cryptosystem deployed in the public domain. Classical cryptographic schemes such as elliptic-curve cryptography [25] and RSA [27] went through decades of testing, analysis, and invention of different physical attacks and their countermeasures to generate enough confidence to be successfully deployed in the real world. In comparison, post-quantum cryptography (PQC),

F. Regazzoni et al. (Eds.): SPACE 2023, LNCS 14412, pp. 162–184, 2024.
https://doi.org/10.1007/978-3-031-51583-5_10

or specifically lattice-based cryptography (LBC) has gone through significantly less amount of investigation in the context of physical attacks. Therefore, although the United States government's National Institute of Standards and Technology (NIST) has recently proposed some standard PQC schemes [1], for a successful transition to PQC, it is imperative that we concentrate our research efforts in this direction.

Masking [11] is an interesting countermeasure against passive physical attacks or side-channel attacks (SCA) such as power analysis, electromagnetic radiation analysis, etc. On a fundamental level, masking works by splitting the secret into multiple random shares and performing the same computation as the unmasked version on each share. Thus, the security of masking is based on the same information-theoretic principles, such as Shamir's secret sharing [29] or multi-party computation [30]. Masking can provide provably secure countermeasures against side-channel attacks. Nevertheless, due to the duplication of computations, the runtime of a masked implementation theoretically grows significantly with the increase in the order of masking. For example, in the case of Kyber, a post-quantum key-encapsulation mechanism (KEM) scheme that has been selected as standard in the NIST's procedure, the runtime of the first, second, and third order of masked implementation is 12, 20, and 30 times of the unmasked implementation on ARM Cortex-M4 platform [10].

Our primary motivation in this work is to assess how the design decisions of a lattice-based KEM scheme, such as the choice of quotient polynomial, distribution of secrets and errors, underlying hard problems, modulus, etc., influence their masking performance. We also want to test how close we can get to the theoretical upper bound of efficiency in masking. For our experiments, we have chosen the post-quantum KEM suite Scabbard [5] with 3 different lattice-based schemes. First, a ring-learning with rounding (RLWR) based scheme Florete with ring size comparable to NewHope [2], second a module-learning with rounding (MLWR) based scheme Sable with ring size similar to Saber [15] and Kyber [8], and finally an MLWR-based scheme Espada with unique smaller ring size. The choice of Scabbard helps us to demonstrate our methods on diverse KEM schemes with many variations in the design. Scabbard was proposed to improve the NIST PQC finalist KEM Saber [15]. The designers of Scabbard argued that all the design decisions of Scabbard had been propelled by the experience gained in the research and developments in the field of lattice-based cryptography of previous years. Therefore, it inherits all the advantages of Saber *i.e.* less randomness due to rounding, power-of-two modulus for efficient masking, simple algorithms for efficiency and faster deployment on diverse platforms, etc. Further, the design of Scabbard improves in areas like suitability for parallel implementation, flexibility, efficiency, and adaptation of faster masking schemes. We will discuss the schemes of Scabbard in Sect. 2.1. In the original publication [5], the authors have provided different implementations on hardware and software platforms to prove their claims on efficiency. It was shown before that the design of Saber is highly conducive to masking [4]. Due to these reasons, Scabbard is an ideal choice to

demonstrate the interplay between design choices and masking performance in lattice-based KEMs.

In this work, we propose arbitrary-order masked implementations of all the KEMs in the suite Scabbard. We implement and benchmark them on an ARM Cortex-M4 microcontroller platform using the PQM4 [21] library to prove the masking friendliness of its design. The ring size of the polynomial length matches the number of message bits, which is 256 for Saber or Kyber as well as Sable. So, the encoding of message bits to the ciphertext polynomial is trivial in these cases. However, this is not the case for Florete and Espada, and these schemes use `original_msg` function for message decoding and `arrange_msg` function for message encoding. This work introduces a higher-order masked version of `original_msg` and `arrange_msg` function. These functions can be applied to all LWR-based KEMs with different ring sizes than 256 and even learning with errors (LWE) based KEMs with some modifications. The schemes of Scabbard use different centered binomial distributions compared to Saber or Kyber. For this purpose, we modified the masked centered binomial distribution (CBD) algorithms proposed by Schneider et al. [28] for each scheme of Scabbard and optimized it for them. Public and re-encrypted ciphertext comparison is an important part of the Fujisaki-Okamoto transformation used in LWE-/LWR-based KEM. It is faster for unmasked or first-order masking but becomes computationally expensive for higher-order maskings. Here, we modified the ciphertext comparator of [23] for each scheme of Scabbard to obtain better performance. These masked components are faster in Scabbard than Kyber, thanks to the choice of RLWR/MLWR hard problem, power-of-two moduli and slightly reduced parameter sets.

As performance results, the overhead factor we obtained for masked Florete for the first-, second-, and third-order are approximately 2.7×, 5×, and 7.7×, compared to the unmasked implementation. For Espada, the overhead cost of the first-, second-, and third-order masked versions are roughly 1.8×, 2.8×, and 4× than the unmasked one. The performance cost of masked Sable for the first-, second-, and third-order are around 2.4×, 4.3×, and 6.3× over the unmasked version. We compare the masked implementations of Florete, Espada, and Sable with the state-of-the-art masked implementation of Kyber and Saber. We show that the masked implementations of all the schemes of Scabbard surpass the masked implementations of Kyber in terms of performance for any order masking, and masked implementations of Florete and Sable outperform masked implementations of Saber for arbitrary order. More specifically, masked Florete performs 73%, 71%, and 70% better than masked Kyber, corresponding to the first-, second-, and third-order. Espada shows 56%, 59%, and 60% performance improvement for first-, second- and third-order masked implementations compared to Kyber. Masked Sable exceeds the execution time of masked Kyber by 75%, 74%, and 73% for the first-, second-, and third-order. Our masked implementations are available at https://github.com/Suparna-Kundu/Masked_Scabbard.git.

To conclude this section, we want to draw attention to the fact that although the NIST standardization procedure for PKE/KEM has been finalized with Kyber, we firmly believe that further investigations and innovations are required to improve side-channel secure PQC schemes. The NIST procedure opened the possibility of exploring different possibilities to improve various aspects of PQC schemes. We have witnessed this throughout the course and even after the NIST procedure. For example, Mitaka [16] has been proposed, which is a masking-friendly version of Falcon [17], a NIST standard for digital signatures. Kyber-90s version of Kyber was proposed to use the advanced encryption standard (AES) as a pseudo-random number generator instead of the slower Keccak extended output function. Similarly, Saber-90s and uSaber were proposed as alternate versions of the NIST PQC standardization finalist scheme Saber to improve efficiency and ease of masking. As discussed earlier, Scabbard [5] was an improvement of Saber. The design of Scabbard has further influenced the design of PQC KEM Smaug [12], which is a candidate scheme from ongoing Korean PQC standardization [22]. Therefore, exploring various design choices and their effect on different aspects of the performance of existing PQC schemes is an interesting research direction.

2 Preliminaries

For a positive integer q, the set of integers modulo q is denoted by $\mathbb{Z}_q$. The quotient ring $\mathbb{Z}_q[x]/f(x)$ is denoted by $\mathcal{R}_q^n$, where $f(x)$ is a n degree cyclotomic polynomial over $\mathbb{Z}_q[x]$. We use lowercase letters to denote an element of this ring, which is a polynomial. We indicate the ring of l length vectors over the ring $\mathcal{R}_q^n$ as $(\mathcal{R}_q^n)^l$ and use bold lowercase letters to denote an element of this ring which is a vector of polynomials. The ring of $l \times l$ length matrices over the ring $\mathcal{R}_q^n$ as $(\mathcal{R}_q^n)^{l\times l}$. The elements of this ring are $l \times l$ matrices of polynomials and are represented by uppercase letters. $x \leftarrow \chi(S)$ represents that x is sampled from the set S and follows the distribution χ. When x is generated using a pseudo-random number generator expanding a seed $seed_x$ over the set S, we denote it as $x \leftarrow \chi(S; seed_x)$. We use $\mathcal{U}$ to denote the uniform distribution and the CBD whose standard deviation $\sqrt{\mu/2}$ is presented by β_μ. We denote the rounding operator with $\lfloor \cdot \rceil$, which returns the closest integer and is rounded upwards during ties. These operations can be extended over the polynomials by applying them coefficient-wise. The polynomial multiplication between two polynomials of length n is represented using $n \times n$ multiplication. We use $\{x_i\}_{0\leq i\leq t}$ to represent the set $\{x_0,\ x_1,\ \ldots,\ x_t\}$ which contains $t+1$ elements of the ring $\mathcal{R}$.

2.1 Scabbard: A Post-quantum KEM Suite

Scabbard is a suite of post-quantum KEMs proposed by Mera et al. [5] that improved state-of-the-art LBC schemes by incorporating different design choices and newer developments in the field. The security of the schemes in the Scabbard depends on some variants of learning with rounding (LWR) problems, more

specifically, module-LWR (MLWR) and ring-LWR (RLWR) problems. Banerjee et al. [3] introduced the LWR problem and also showed that the LWR problem is as hard as the LWE problem. If $A \leftarrow \mathcal{U}((\mathbb{Z}_q)^{l\times l})$, secret $\mathbf{s} \leftarrow \beta_\mu((\mathbb{Z}_q)^l)$, error $\mathbf{e} \leftarrow \beta_{\mu_e}((\mathbb{Z}_q)^l)$, and $\mathbf{b} \leftarrow \mathcal{U}((\mathbb{Z}_q)^l)$ then distinguishing between $(A,\ As+e)$ and $(A,\ b)$ is hard and this problem is known as the decision version of LWE problem. The decision version of the LWR problem states that if $A \leftarrow \mathcal{U}((\mathbb{Z}_q)^{l\times l})$, secret $\mathbf{s} \leftarrow \beta_\mu((\mathbb{Z}_q)^l)$, and for some $p < q$, $\mathbf{b} \leftarrow \mathcal{U}((\mathbb{Z}_p)^l)$ then distinguishing between $(A,\ \lfloor(q/p)As\rceil)$ and $(A,\ b)$ is hard [3]. In the LWR problem, the explicit sampling of error $\mathbf{e}$ in the LWE is replaced by the rounding operation. In case of the MLWR problem, $A \leftarrow \mathcal{U}((\mathcal{R}_q^n)^{l\times l})$, $\mathbf{s} \leftarrow \beta_\mu((\mathcal{R}_q^n)^l)$, $\mathbf{b} \leftarrow \mathcal{U}((\mathcal{R}_p^n)^l)$ and the MLWR problem states that $(A,\ \lfloor(q/p)As\rceil)$ and $(A,\ b)$ are computationally indistinguishable [24]. In standard LWR-based and RLWR-based constructions, the ranks of underlying matrices are respectively l and n, with very high probability. On the other hand, MLWR-based constructions are proposed as a trade-off between standard LWR-based and RLWR-based structures. The rank of underlying matrices in MLWR-based schemes is $l \times n$. It makes the structures of MLWR-based constructions more generic, as we can convert the MLWR-based scheme to a standard LWR-based one by fixing $n = 1$ and an RLWR-based one by setting $l = 1$. Therefore, we use MLWR notations to describe the schemes in Scabbard below. A KEM needs to be secure against chosen ciphertext attacks (IND-CCA/IND-CCA2: indistinguishable against a-posteriori chosen-ciphertext attacks). In LWR-based KEM, it is accomplished by applying Jiang *et al.*'s version [20] of Fujisaki-Okamoto (FO) transformation [18] over the generic LWR-based public-key encryption (PKE), where the PKE needs to be secure against chosen plaintext attacks (IND-CPA: indistinguishable against chosen plaintext attack). We denote generic LWR-based PKE as `LWR.PKE` and generic LWR-based KEM as `LWR.KEM`, which are shown respectively in Fig. 1 and Fig. 2. In `LWR.KEM`, $\mathcal{H}$, $\mathcal{G}$, and `KDF` three hash functions are required as part of FO transformation. This suite of KEMs consists of three schemes: (i) Florete, (ii) Espada, and (iii) Sable. We briefly describe these three schemes with their specific features below.

`LWR.PKE.KeyGen()`

1. $seed_{\boldsymbol{A}} \leftarrow \mathcal{U}(\{0,1\}^{256})$
2. $\boldsymbol{A} \leftarrow \mathcal{U}((\mathcal{R}_q^n)^{l\times l}; \text{seed}_{\boldsymbol{A}})$
3. $r \leftarrow \mathcal{U}(\{0,1\}^{256})$
4. $\boldsymbol{s} \leftarrow \beta_\mu((\mathcal{R}_q^n)^l; r)$
5. $\boldsymbol{b} = ((\boldsymbol{A}^T\boldsymbol{s}+\boldsymbol{h}) \bmod q) \gg (\epsilon_q-\epsilon_p) \in (\mathcal{R}_p^n)^l$
6. **return** $(pk = (seed_{\boldsymbol{A}}, \boldsymbol{b}),\ sk = (\boldsymbol{s}))$

`LWR.PKE.Enc`$(pk = (seed_{\boldsymbol{A}}, \boldsymbol{b}), m \in R_2; r)$

1. $\boldsymbol{A} \leftarrow \mathcal{U}((\mathcal{R}_q^n)^{l\times l}; \text{seed}_{\boldsymbol{A}})$
2. **if:** r is not specified:
3. $\quad r \leftarrow \mathcal{U}(\{0,1\}^{256})$
4. $\boldsymbol{s'} \leftarrow \beta_\mu((\mathcal{R}_q^n)^l; r)$
5. $\boldsymbol{u} = ((\boldsymbol{A}\boldsymbol{s'}+\boldsymbol{h}) \bmod q) \gg (\epsilon_q-\epsilon_p) \in (\mathcal{R}_p^n)^l$
6. $c_m = \boldsymbol{b}^T(\boldsymbol{s'} \bmod p) \in \mathcal{R}_p^n$
7. $v = (c_m + h_1 - 2^{\epsilon_p-B} m \bmod p) \gg (\epsilon_p-\epsilon_t-B) \in \mathcal{R}_{2^B t}^n$
8. **return** $c = (\boldsymbol{u}, v)$

`LWR.PKE.Dec`$(sk = \boldsymbol{s}, c = (\boldsymbol{u}, v))$

1. $u'' = \boldsymbol{u}^T(\boldsymbol{s} \bmod p) \in \mathcal{R}_p^n$
2. $m'' = (u'' - 2^{\epsilon_p-\epsilon_t-B} v + h_2) \bmod p$
3. $m' = m'' \gg (\epsilon_p - B) \in \mathcal{R}_{2^B t}^n$
4. **return** m'

Fig. 1. Generic `LWR.PKE` [5]

LWR.KEM.KeyGen()

1. $(seed_{\boldsymbol{A}}, \boldsymbol{b}, \boldsymbol{s}) = \texttt{LWR.PKE.KeyGen}()$
2. $pk = (seed_{\boldsymbol{A}}, \boldsymbol{b})$
3. $pkh = \mathcal{H}(pk)$
4. $z \leftarrow \mathcal{U}(\{0,1\}^{256})$
5. **return** $(pk = (seed_{\boldsymbol{A}}, \boldsymbol{b}), sk = (\boldsymbol{s}, z, pkh))$

LWR.KEM.Encaps$(pk = (seed_{\boldsymbol{A}}, \boldsymbol{b}))$

1. $m' \leftarrow \mathcal{U}(\{0,1\}^{256})$
2. $m = \texttt{arrange\_msg}(m')$
3. $m = \mathcal{H}(m)$
4. $(\hat{K}, r) = \mathcal{G}(\mathcal{H}(pk), m)$
5. $c = \texttt{LWR.PKE.Enc}(pk, m; r)$
6. $K = \texttt{KDF}(\hat{K}, \mathcal{H}(c))$
7. **return** (c, K)

LWR.KEM.Decaps$(sk = (\boldsymbol{s}, z, pkh), pk = (seed_{\boldsymbol{A}}, \boldsymbol{b}), c)$

1. $m'' = \texttt{LWR.PKE.Dec}(\boldsymbol{s}, c)$
2. $m' = \texttt{original\_msg}(m'')$
3. $(\hat{K}', r') = \mathcal{G}(pkh, m')$
4. $c_* = \texttt{LWR.PKE.Enc}(pk, m'; r')$
5. **if:** $c = c_*$
6. **return** $K = \texttt{KDF}(\hat{K}', \mathcal{H}(c))$
7. **else:**
8. **return** $K = \texttt{KDF}(z, \mathcal{H}(c))$

Fig. 2. Generic LWR.KEM [5]

2.1.1 Florete This scheme is based on the RLWR problem *i.e.* $l = 1$ in Fig. 1 and designed for faster running time. Here, the cyclotomic polynomial used to construct the quotient rings $\mathcal{R}_q^n$, $\mathcal{R}_p^n$, and $\mathcal{R}_t^n$ is $(x^{768} - x^{384} + 1)$. In Florete, one message bit is encoded in three coefficients of the polynomial v in line 7 of LWR.PKE.Enc algorithm of Fig. 1. So, during the encapsulation process, as shown in line 2 of LWR.KEM.Encaps algorithm of Fig. 2, a conversion from 256 bits of message to a polynomial of length 768 is performed with the help of arrange_msg function and it is defined as: $\texttt{arrange\_msg}(m') = m'||m'||m'$. The inverse of arrange_msg function is used in the LWR.KEM.Decaps algorithm named as original_msg, and the $\texttt{original\_msg} : \mathbb{Z}_2^{768} \longrightarrow \mathbb{Z}_2^{256}$ is defined as if $\texttt{original\_msg}(m'') = m'$ and $b \in \{0, 1, \ldots, 255\}$ then $m'[b] = \begin{cases} 0 & \text{if } m''[b] + m''[b+256] + m''[b+512] \leq 1 \\ 1 & \text{otherwise} \end{cases}$. In Florete, 768×768 polynomial multiplication is used, and it is performed using the combination of Toom-Cook 3-way, Toom-Cook 4-way, 2 levels of Karatsuba, and 16×16 schoolbook multiplication.

2.1.2 Espada This scheme is designed to reduce the memory footprint on software platforms. It is based on the MLWR problem, and the cyclotomic polynomial is used to construct the underlying quotient ring of the lattice problem $\mathcal{R}_q^n$ is $(x^{64} + 1)$. The polynomial length here is 64, so the dimension of vectors of polynomial l is taken equal to 12 to maintain security. In Espada, the 256 bit message is encoded inside the 64 length polynomial v, so four message bits are encoded in a coefficient of the polynomial v. The $\texttt{arrange\_msg} : \mathbb{Z}_2^{256} \longrightarrow \mathbb{Z}_4^{64}$ and

the function is defined as: $\texttt{arrange\_msg}(m') = m''$, where for $b \in \{0,\ 1,\ \ldots,\ 63\}$

$$m''[b] = m'[4 * b + 3]||m'[4 * b + 2]||m'[4 * b + 1]||m'[4 * b]. \tag{1}$$

The $\texttt{original\_msg} : \mathbb{Z}_4^{64} \longrightarrow \mathbb{Z}_2^{256}$ function is defined as: $\texttt{original\_msg}(m'') = m'$ and follows Eq. 1. Lastly, the 64×64 polynomial multiplication is performed using 2 levels of Karatsuba and 16×16 schoolbook multiplication.

2.1.3 Sable This scheme can be interpreted as an alternate version of Saber and is designed to improve performance with less memory footprint. It is also based on the MLWR problem, and similar to Saber, the cyclotomic polynomial used here in the quotient rings is $(x^{256} + 1)$. The `arrange_msg` function and `original_msg` function are described as: $\texttt{arrange\_msg}(m') = m'$ and $\texttt{original\_msg}(m'') = m'' = m'$, respectively. The polynomial multiplication used in Sable is identical to Saber. The 256×256 polynomial multiplication is realized by the combination of Toom-Cook 4-way, 2 levels of Karatsuba, and 16×16 schoolbook multiplication.

The concrete security of these schemes depends on the parameter set, which includes the three power-of-two ring moduli $t < p < q$, the length of a polynomial n, the dimension of the vector of polynomial l, the CBD parameter μ, and the number of message-bit encoded in a coefficient of the polynomial is represented by B. Table 1 presents the parameter sets for all three schemes that achieve the NIST security level 3. We humbly refer to the original Scabbard paper [5] for more insightful details.

Table 1. Parameters of Scabbard suite

Scheme Name	Ring/Module Parameters		PQ Security	Failure probability	Moduli		CBD (β_η)	Encoding	Key sizes for KEM (Bytes)	
Florete	n:	768	2^{157}	2^{-131}	ϵ_q:	10	$\eta = 1$	B = 1	Public key:	896
					ϵ_p:	9			Secret key:	1152
	l:	1			ϵ_t:	3			Ciphertext:	1248
Espada	n:	64	2^{128}	2^{-167}	ϵ_q:	15	$\eta = 3$	B = 4	Public key:	1280
					ϵ_p:	13			Secret key:	1728
	l:	12			ϵ_t:	3			Ciphertext:	1304
Sable	n:	256	2^{169}	2^{-143}	ϵ_q:	11	$\eta = 1$	B = 1	Public key:	896
					ϵ_p:	9			Secret key:	1152
	l:	3			ϵ_t:	4			Ciphertext:	1024

2.2 Masking

The effectiveness of masking against SCA has been well demonstrated for symmetric-key block ciphers [13,26] and recently extended for LBC [4,9,23]. In n-th order masking, we split the sensitive data x into $(n+1)$ shares and perform all the operations on each share separately. So, an adversary with a limited number of probes, such as at most n probes, does not receive any advantages compared to another adversary who does not have access to those probes. The nth order masking technique can prevent up to nth order differential power attacks. However, the integration of masking techniques in LBC schemes affects the performance of the algorithm significantly with the increment of the masking order. The design decision of cryptographic schemes affects the performance of masked versions of the lattice-based schemes. This is why even though the unmasked performance of NIST finalist Saber is almost the same as Kyber, the masked version of Saber is way faster than masked Kyber for any masking order. Masked version Saber gains this advantage thanks to the choice of LWR problem and power-of-two moduli. The KEMs in the suite Scabbard also use power-of-two moduli and further improve the efficiency of the LWR-based schemes. In this work, we investigate whether the efficiency of Scabbard will translate to the masked domain.

3 Masking Scabbard

The CCA-secure KEM schemes are used to share secrets among communicating parties. Here, the secret key is non-ephemeral *i.e.* the key generation is run once to generate a long-term secret key that can be used for multiple sessions and communicating with multiple entities. Therefore, in a KEM scheme, only the decapsulation is executed multiple times to retrieve the secret data from multiple entities through multiple sessions. However, this is also advantageous for an adversary. The adversary can run the decapsulation operation multiple times to improve the precision of its fault injection or take multiple side-channel traces to reduce noise in its measurements, thus improving its success probability. Mounting attacks on other operations, such as key generation and encapsulation, are relatively harder. Once an adversary compromises the secret key, it can use it to expose the secret keys of multiple sessions. Therefore, protecting the decapsulation operation from side-channel attacks is critical for the side-channel security of a KEM. We display the flow of the decapsulation algorithm of generic LWR-based KEM in Fig. 3 and denoted vulnerable operations in the color gray. Here `original_msg` and `arrange_msg` functions are shown by `OMsg` and `AMsg`. In this section, we will describe the masking methods of all the components susceptible to SCA in the decapsulation operation of the Scabbard schemes.

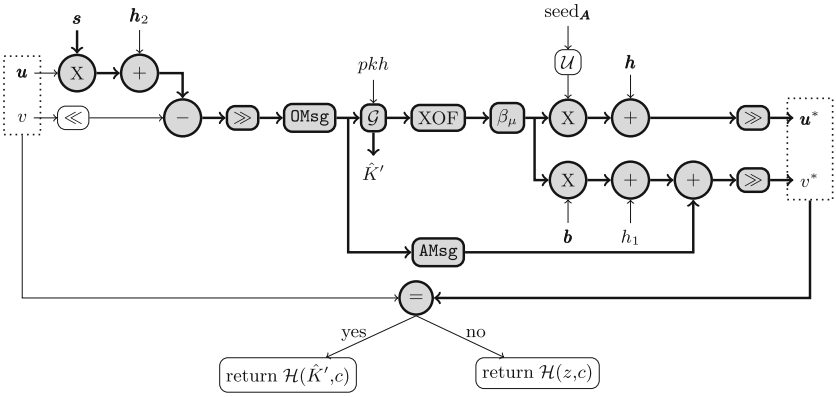

Fig. 3. Decapsulation of LWR-based KEM. The operations in color gray are involved with the long-term secret $\boldsymbol{s}$ and are susceptible to side-channel attacks

Here, we have used two masking techniques: (i) arithmetic masking and (ii) Boolean masking to mask the Scabbard suite's schemes because these schemes consist of some operations that are cheaper to mask using arithmetic masking and some are easy to mask using Boolean masking. In both the t-order arithmetic and Boolean masking techniques, first we split the sensitive operand $x \in \mathbb{Z}_q = \mathbb{Z}_{2^{\epsilon_q}} = \mathbb{Z}_2^{\epsilon_q}$ into $(t+1)$ shares, such as $x_0, x_1, \ldots, x_t \in \mathbb{Z}_q$. However, for arithmetic masking the relation between x and $(t+1)$ shares of x is $x = (x_0 + x_1 + \cdots + x_t) \bmod q$, and in Boolean masking the relation between x and $(t+1)$ shares of x is $x = (x_0 \oplus x_1 \oplus \cdots \oplus x_t)$.

3.1 Arithmetic Operations

It can be seen from Fig. 3 that the decapsulation algorithm of each KEM of the suite Scabbard consists of mostly arithmetic operations, such as polynomial multiplications, polynomial addition, and polynomial subtractions. These operations can be masked efficiently utilizing arithmetic masking. Here, we need to duplicate these operations for each arithmetic share and perform them separately. The performance cost of these operations grows linearly with the increase of arithmetic shares.

Although this part is more or less similar for all the LWE/LWR-based KEMs (for example, Kyber and Saber), the parameter set impacts the performance of unmasked and masked versions of these operations. This also helps the schemes of Scabbard to achieve better performance compared to other LBC-based KEMs in some scenarios. The performance cost of the masked arithmetic operations in Sable is less than Saber or Kyber because the total cost of arithmetic operations of Sable is less than Saber or Kyber in the unmasked domain. It happens because Sable uses a slightly reduced parameter set than Saber. However, the performance cost of arithmetic operations in Florete or Espada is more than Saber or Kyber, as is the case in the unmasked domain.

3.2 Compression

Compression operation is the final step of the `LWR.PKE.Dec` algorithm, and in this step, encoded message bits are retrieved from the polynomial m'' after performing the reconciliation. For Florete and Sable, only the most significant bit is extracted, and for Espada, the four most significant bits are extracted from each coefficient of the polynomial m''. After that, these message bits are used as input in `SHA3-512` hash function for computing the seed s' for the re-encryption procedure. These message bits are also needed to construct the session key. The extraction of the most significant bits is performed by using a logical shift operation in LWR-based KEM. This operation is easy to protect with Boolean masking. However, in the masked setting, the input of the compression operation is arithmetically masked, as its previous steps consisted of arithmetic operations. So, in the masked compression operation, first, we apply arithmetic to Boolean (`A2B`) conversion, and then we perform coefficient-wise $\epsilon_p - B$ bit right shift operation [23].

This compress operation in Sable is very similar to the one used in Saber, except for the value of ϵ_p. The value of the parameter ϵ_p is smaller in Sable than in Saber. So, the performance of `A2B` conversion is relatively better in Sable compared to Saber. Hence, the overall performance of the masked compress operation is better in Sable than in Saber. The compress operation of Florete is also similar to the compress operation used in Saber. The value of parameters ϵ_p in Florete is the same as Sable and so a little smaller than in Saber. However, the degree of the message containing part of the ciphertext polynomial is 768 in Florete, while it is 256 in Saber. So, the number of coefficients in Florete is three times compared to Saber. The performance cost of `A2B` conversion and $\epsilon_p - 1$ right shift operation in Florete is approximately three times the performance cost of these operations in Saber. Therefore, the performance of the masked compress operation in Florete takes approximately three times the cycles compared to the masked compress operation in Saber. The scheme Espada encodes four message bits in a single coefficient of ciphertext, and the polynomial size in Espada is 64, which is 1/4th of the polynomial size in Saber. The value of ϵ_p in Espada is slightly bigger than in Saber. However, the `A2B` conversion component is faster in Espada than in Saber due to the small polynomial size. Also, for the same reason, the coefficient-wise $\epsilon_p - 4$ bit right shift operation in Espada is faster than the coefficient-wise $\epsilon_p - 1$ bit right shift operation of Saber. Overall, the performance of the masked compress operation of Espada is roughly four times faster compared to the masked compress operation in Saber. As Kyber uses prime moduli, the masked compress operation of Kyber is far more complicated and has some extra steps. These extra steps includes conversion of arithmetic shares from $\mathbb{Z}_q$ to power-of-two modulus $\mathbb{Z}_{2^{k_q}}$, where $\log\ q < 2^{k_q}$. These are computationally quite expensive operations. Due to the power-of-two moduli, schemes in Scabbard and Saber do not need these additional steps. This results in more efficient masked compress operation for these schemes.

3.3 Message Decoding and Encoding

For Florete and Espada, the bit length of the message *i.e* 256 is not equal to the sizes of the polynomial ring, which are 768 and 64, respectively. Authors of Scabbard proposed techniques to encode and decode the message into the polynomial named `arrange_msg` and `original_msg` respectively. The encoding and decoding operation where the polynomial ring length is the same as the message length is very straightforward, and we do not need any special masking gadget for `original_msg` and `arrange_msg` functions. However, we need to use a special masking component to mask the `original_msg` function when polynomial length equals r times message bits, where $r > 1$, e.g., Florete, NewHope [2]. We use r coefficients to hide one message bit in this case. We also have to use a special masking gadget to mask the `arrange_msg` function if the number of message bits equals B times a polynomial length, where $B > 1$, e.g., Espada. In these schemes, B message bits are hidden in a coefficient. We discuss these gadgets below.

Message Decoding: In Florete, 3 coefficients had been used to hide one message bit. The $\texttt{original\_msg} : \mathbb{Z}_2^{768} \longrightarrow \mathbb{Z}_2^{256}$ is defined here as if $\texttt{original\_msg}(m'') = m'$ and $b \in \{0,\ 1,\ \ldots,\ 255\}$ then $m'[b] = \begin{cases} 0 & \text{if } m''[b] + m''[b+256] + m''[b+512] \leq 1 \\ 1 & \text{otherwise} \end{cases}$.
First, we perform secure additions (`SecAdd`) over Boolean shared data to mask this function, and the possible output must be one of $\{0,\ 1,\ 2,\ 3\}$. Notice that it is always a two-bit number for any bit b. The output of the `original_msg` is equal to the most significant bit, which is the 2nd bit. So, after performing the masked addition, we extract the most significant bit of the masked output shares (2nd bit). At last, we return the most significant bit as output `original_msg` for each bit $b \in \{0,\ 1,\ \ldots,\ 255\}$. We present this masked function in Algorithm 1.

Algorithm 1: Masked `original_msg` function for Florete

Input : $\{m_i''\}_{1 \leq i \leq n}$ where $m_i'' \in \mathbb{Z}_2^{768}$ such that $\bigoplus_{i=1}^{n} m_i'' = m''$
Output : $\{m_i'\}_{1 \leq i \leq n}$ where $m_i' \in \mathbb{Z}_2^{256}$, $\bigoplus_{i=1}^{n} m_i' = m'$ and $\texttt{original\_msg}(m'') = m'$

1 **for** *j=0 to 255* **do**
2 $\quad \{x_i[j]\}_{1 \leq i \leq n} \leftarrow m_i''[j]$; $\{y_i[j]\}_{1 \leq i \leq n} \leftarrow m_i''[256 + j]$;
$\quad \{z_i[j]\}_{1 \leq i \leq n} \leftarrow m_i''[512 + j]$
3 $\{w_i\}_{1 \leq i \leq n} \leftarrow \texttt{SecAdd}(\{x_i\}_{1 \leq i \leq n}, \{y_i\}_{1 \leq i \leq n})$
4 $\{w_i'\}_{1 \leq i \leq n} \leftarrow \texttt{SecAdd}(\{w_i\}_{1 \leq i \leq n}, \{z_i\}_{1 \leq i \leq n})$
5 $\{m_i'\}_{1 \leq i \leq n} \leftarrow \{w_i'\}_{1 \leq i \leq n} \gg 1$
6 **return** $\{m_i'\}_{1 \leq i \leq n}$

Message Encoding: In Florete and Sable, a co-efficient of the message polynomial carries a single message bit. Here, `arrange_msg` is defined by `arrange_msg` : $\mathbb{Z}_2^{256} \longrightarrow \mathbb{Z}_2^{768}$ and `arrange_msg` : $\mathbb{Z}_2^{256} \longrightarrow \mathbb{Z}_2^{256}$ for Florete and Sable respectively. The Boolean masked output of this function then takes part in the modular addition in the next step of the re-encryption stage as the message polynomial. As the shares of each coefficient of the message polynomial are in $\mathbb{Z}_2$, the Boolean shares are equivalent to the arithmetic shares. Hence, we can skip the Boolean to arithmetic conversion here. However, for Espada, we encode four message bits in a single co-efficient of the message polynomial, and `arrange_msg` is defined by `arrange_msg` : $\mathbb{Z}_2^{256} \longrightarrow \mathbb{Z}_4^{64}$. So, we need to convert Boolean shares of each coefficient of message polynomial to arithmetic shares using the `B2A` algorithm. After that, we perform the modular addition with two arithmetically masked inputs.

3.4 Hash Functions

Decapsulation algorithm uses one hash functions G (`SHA3-512`) and one pseudo-random number generator XOF (`SHAKE-128`). These functions are different instances of the sponge function Keccak-f[1600] [6]. It consists of five steps: (i) θ, (ii) ρ, (iii) π, (iv) χ, and (v) ι. Among the five steps, θ, ρ, and π are linear diffusion steps and ι is a simple addition. As all these four steps are linear operations over Boolean shares, in masked settings, we repeat all these operations on each share separately. Only χ is a degree 2 non-linear mapping and thus requires extra attention to mask. Overall, Keccak-f[1600] is less expensive to mask by using Boolean masking. Here, we use the higher-order masked Keccak proposed by Gross et al. [19]. Due to the compact parameter choices, Scabbard schemes require fewer pseudo-random numbers than Saber. Eventually, this leads to fewer invocations of the sponge function Keccak in Florete and Sable than in Espada. Moreover, the output length of `SHAKE-128` is the same for Florete and Sable, which is even smaller than Espada. To sum up, the performance cost of the masked XOF `SHAKE-128` is lower in Florete, Sable, and Espada compared to Saber.

3.5 Centered Binomial Sampler

The re-encryption part of the decapsulation algorithm contains a centered binomial sampler for sampling the vector $\mathbf{s}'$. This sampler outputs $\mathtt{HW}(x) - \mathtt{HW}(y)$, where x and y are pseudo-random numbers and `HW` represents hamming weight. The bit size of pseudo-random numbers x and y depends on the scheme. These pseudo-random numbers are produced employing `SHAKE-128`. As mentioned in the previous section, these function is efficient if we mask with the help of Boolean masking. Hence, the shares generated from `SHAKE-128` are Boolean. However, upon constructing the $\mathbf{s}'$, we need to perform modular multiplication with inputs $\mathbf{s}'$ and public-key $\mathbf{b}$. This is efficient if we use arithmetic masking. Therefore, we need to perform Boolean to arithmetic conversion in the masked-centered binomial sampler. Schneider et al. [28] proposed two centered

binomial samplers, $\texttt{Sampler}_1$ and $\texttt{Sampler}_2$. $\texttt{Sampler}_1$ first converts Boolean shares of x and y to arithmetic shares then computes $\texttt{HW}(x) - \texttt{HW}(y)$ by using arithmetic masking technique. $\texttt{Sampler}_2$ first computes $z = \texttt{HW}(x) - \texttt{HW}(y) + k$, where $k \geq \mu/2$ using Boolean masking. After that, it converts Boolean shares of z to arithmetic shares and then performs $z - k$ using the arithmetic masking technique to remain with arithmetic shares of $\texttt{HW}(x) - \texttt{HW}(y)$. $\texttt{Sampler}_1$ uses a bit-wise masking procedure, while $\texttt{sampler}_2$ uses the bitslicing technique on some parts of the algorithm for receiving better throughput. We have adopted these two samplers and optimized them to mask the CBD function of each KEM of the Scabbard suite. We could not directly use the optimized CBD used in Saber [23], as that one is optimized for β_8, and schemes of Scabbard use smaller CBD to sample the vector $\mathbf{s}'$. Schemes like Kyber and NewHope [2,28] use prime modulus. So, a few components there are different, for example, the B2A conversion and extra modular addition. As Scabbard uses power-of-two moduli, these components can be implemented in a much cheaper way for them. We describe the optimized masked CBD samplers for these schemes below.

3.5.1 Florete and Sable In these two schemes, we take advantage of the centered binomial sampler with a small standard deviation, β_2. For β_2, x and y are 1-bit pseudo-random numbers. We have adopted $\texttt{Sampler}_1$ and $\texttt{Sampler}_2$, with these specification. As $\texttt{Sampler}_2$ is designed to provide a better performance, we started with the adaptation of $\texttt{Sampler}_2$ for β_2 named $\texttt{MaskCBDSampler}_A$ as shown in Algorithm 2. In this algorithm, first, we perform SecBitSub on Boolean shares of x and y to calculate Boolean shares of $\texttt{HW}(x) - \texttt{HW}(y)$. Second, we add constant 1 with the output shares of SecBitSub to avoid negative numbers. Third, we convert the output from Boolean shares to arithmetic shares with the help of the B2A conversion algorithm proposed in [7]. In the last step, we subtract the added constant in step-2, which converts secret shares from $\{0, 1, 2\}$ to $\{-1, 0, 1\}$.

Algorithm 2: $\texttt{MaskCBDSampler}_A$ ([28], using $\texttt{sampler}_2$)

Input : $\{x_i\}_{0\leq i\leq n}, \{y_i\}_{0\leq i\leq n}$ where $x_i, y_i \in \mathbb{R}_2$ such that $\bigoplus_{i=0}^{n} x_i = x, \bigoplus_{i=0}^{n} y_i = y$
Output : $\{A_i\}_{0\leq i\leq n}$ where $A_i \in \mathbb{R}_q$ and $\sum_{i=0}^{n} A_i = (\texttt{HW}(x) - \texttt{HW}(y)) \bmod q$

1 $\{z_i\}_{0\leq i\leq n} \leftarrow \texttt{SecBitSub}(\{x_i\}_{0\leq i\leq n}, \{y_i\}_{0\leq i\leq n})$
2 $z_0[0] \leftarrow z_0[0] \oplus 1$
3 $\{A_i\}_{0\leq i\leq n} \leftarrow \texttt{B2A}(\{z_i\}_{0\leq i\leq n})$ [7]
4 $A_1 \leftarrow (A_1 - 1) \bmod q$
5 **return** $\{A_i\}_{0\leq i\leq n}$

Algorithm 3: $\texttt{MaskCBDSampler}_B$ ([28], using $\texttt{sampler}_1$)

Input : $\{x_i\}_{0\le i\le n}, \{y_i\}_{0\le i\le n}$ where $x_i, y_i \in \mathbb{R}_2$ 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 $\{T1_i\}_{0\le i\le n} \leftarrow \texttt{B2A}(\{x_i\}_{0\le i\le n})$ [7]; $\{T2_i\}_{0\le i\le n} \leftarrow \texttt{B2A}(\{y_i\}_{0\le i\le n})$ [7]
2 **for** *i=0 to n* **do**
3 $\quad A_i \leftarrow (T1_i - T2_i)$
4 **return** $\{A_i\}_{0\le i\le n}$

Algorithm 4: $\texttt{MaskCBDSampler}_C$ ([28], using $\texttt{sampler}_2$)

Input : $\{x_i\}_{0\le i\le n}, \{y_i\}_{0\le i\le n}$ where $x_i, y_i \in \mathbb{R}_2^3$ 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 $\{z_i\}_{0\le i\le n} \leftarrow \texttt{SecBitAdd}(\{x_i\}_{0\le i\le n})$ [4]
2 $\{z_i\}_{0\le i\le n} \leftarrow \texttt{SecBitSub}(\{z_i\}_{0\le i\le n}, \{y_i\}_{0\le i\le n})$ [28]
3 **for** *i=0 to n* **do**
4 $\quad z_i[2] \leftarrow (z_i[2] \oplus z_i[1])$
5 $z_0[2] \leftarrow z_0[2] \oplus 1$
6 $\{A_i\}_{0\le i\le n} \leftarrow \texttt{B2A}(\{z_i\}_{0\le i\le n})$ [7]
7 $A_1 \leftarrow (A_1 - 4) \bmod q$
8 **return** $\{A_i\}_{0\le i\le n}$

As the bit size of x and y is small for β_2, the bitslice technique for addition and subtraction does not improve the throughput much. So, for comparison purposes, we have adopted the technique of the $\texttt{sampler}_1$ for β_2. We name this algorithm $\texttt{MaskCBDSampler}_A$, and present in Algorithm 3. In this algorithm, we conduct B2A conversions over x and y and then perform share-wise subtraction between arithmetic shares of x and y.

3.5.2 Espada We use the centered binomial sampler, β_6, in this scheme. For β_6, x and y are 3-bit pseudo-random numbers. We have adopted a bitsliced implementation of $\texttt{Sampler}_2$ from [28] for β_6 to achieve better efficiency as the standard deviation of the CBD is large. We name this masked sampler as $\texttt{MaskCBDSampler}_C$, and it is shown in Algorithm 4. Similar to $\texttt{MaskCBDSampler}_B$, $\texttt{MaskCBDSampler}_C$ begins with the SecBitAdd operation, which is performed on Boolean shares of x and generates Boolean shares of $\texttt{HW}(x)$. Then SecBitSub is conducted over the Boolean output shares and Boolean shares of y and outputs Boolean shares of $\texttt{HW}(x) - \texttt{HW}(y)$. After that, the constant 4 is added with the output shares of SecBitSub to avoid negative numbers. In the next step, we convert the output from Boolean shares to arithmetic shares with the help of B2A conversion algorithm proposed in [7]. Finally, we subtract the added constant in step-7 and transform secret shares from $\{1, 2, 3, 4, 5, 6, 7\}$ to $\{-3, -2, -1, 0, 1, 2, 3\}$.

The masked CBD sampler (β_8) used in Saber is faster than the masked CBD of Kyber because of the power-of-two moduli. $\texttt{MaskCBDSampler}_A$ and $\texttt{MaskCBDSampler}_B$ are optimized implementation of β_2, which has been used in Florete and Sable. $\texttt{MaskCBDSampler}_C$ is designed for Espada, which is optimized implementation of β_6. For β_2 and β_6, the B2A conversion is much faster than β_8 thanks to the smaller coefficients size in the input polynomial. Therefore, the performance cost of the masked CBD is less for all the schemes in Scabbard compared to Saber or Kyber. A more detailed performance cost analysis of masked CBD implementations for Scabbard is presented in Sect. 4.1.

3.6 Ciphertext Comparison

It is one of the costliest components for masked implementations of lattice-based KEMs, which is a part of the FO transformation. Previously, many methods have been proposed to perform this component efficiently [9,14,23]. For the masked ciphertext comparison part of each KEM of Scabbard, we have adopted the improved simple masked comparison method used in the higher-order masked implementation of Saber [23]. To the best of our knowledge, this is currently the most efficient masked ciphertext comparison implementation available. Through this process, we compare the arithmetically masked output of the re-encryption component before the right shift operation ($\tilde{u},\ \tilde{v}$) with the unmasked public ciphertext, ($u,\ v$). Additionally, note that $\mathbf{u'} = \tilde{\mathbf{u}} \gg (\epsilon_q - \epsilon_p)$ and $v' = \tilde{v} \gg (\epsilon_p - \epsilon_t - B)$. At first, we perform `A2B` conversion step over the arithmetically masked shares of the output and transform these to Boolean shares, and then we follow the right shift operation. After that, we subtract the unmasked public ciphertext ($u,\ v$) from a share of the Boolean masked output of the `A2B` operation with the help of the XOR operation. Finally, we proceed with checking that all the returned bits of the subtract operation are zero with the `BooleanAllBitsOneTest` algorithm. This algorithm returns 1 only if it receives all the bits encoded in each coefficient of the polynomials is 1; else it returns 0. All these aforementioned steps are presented in Algorithm 5. For further details, we refer to the higher-order masked Saber paper [23].

The parameter settings are different for each KEM of the Scabbard suite. Due to this, byte sizes of the masked inputs of the functions `A2B` and `BooleanAllBitsOneTest` are different for each KEM of the suite, and we show these numbers in Table 2. For reference, we also provide the byte sizes of the masked inputs of `A2B` and `BooleanAllBitsOneTest` for Saber in this table. These differences in the input bytes also affect the performances of corresponding masked implementations. The masked input sizes of both the functions `A2B` and `BooleanAllBitsOneTest` for Sable are less than Saber. On account of this, the performance cost of masked ciphertext comparison is cheaper for Sable than Saber. The masked input sizes of both functions `A2B` and

Algorithm 5: Simple masked comparison algorithm [23]

Input : Arithmetic masked re-encrypted ciphertext $(\{\tilde{\mathbf{u}}_i\}_{0\le i\le n}, \{\tilde{v}_i\}_{0\le i\le n})$ and public ciphertext ($\mathbf{u}$ and v) where each $\tilde{\mathbf{u}}_i \in \mathbb{R}^l_{2^{\epsilon_q}}$ and $\tilde{v}_i \in \mathbb{R}_{2^{\epsilon_p}}$ and $\sum_{i=0}^{n} \tilde{\mathbf{u}}_i \bmod q = \tilde{\mathbf{u}} \sum_{i=0}^{n} \tilde{v}_i \bmod q = \tilde{v}$.

Output : $\{bit\}_{0\le i\le n}$, where with each $bit_i \in \mathbb{Z}_2$ and $\bigoplus_{i=0}^{n} bit_i = 1$ iff $\mathbf{u} = \mathbf{u}' \gg (\epsilon_q - \epsilon_p)$ and $v = v' \gg (\epsilon_p - \epsilon_t - B)$, otherwise 0.

1 $\{\mathbf{y}_i\}_{0\le i\le n} \leftarrow$ `A2B`$(\{\tilde{\mathbf{u}}_i\}_{0\le i\le n})$; $\{x_i\}_{0\le i\le n} \leftarrow$ `A2B`$(\{\tilde{v}_i\}_{0\le i\le n})$

2 $\{\mathbf{y}_i\}_{0\le i\le n} \leftarrow (\{\mathbf{y}_i\}_{0\le i\le n} \gg (\epsilon_q - \epsilon_p))$; $\{x_i\}_{0\le i\le n} \leftarrow (\{x_i\}_{0\le i\le n} \gg (\epsilon_p - \epsilon_t - B))$

3 $\mathbf{y}_1 \leftarrow \mathbf{y}_1 \oplus \mathbf{u}$; $x_1 \leftarrow x_1 \oplus v$

`/* Boolean circuit to test all bits of` $(\mathbf{y}, x)$ `are 0 */`

4 $\mathbf{y}_0 \leftarrow \neg\mathbf{y}_0$; $x_0 \leftarrow \neg x_0$

5 $\{bit_i\}_{0\le i\le n} \leftarrow$ `BooleanAllBitsOneTest` $(\{\mathbf{y}_i\}_{0\le i\le n}, \{x_i\}_{0\le i\le n}, \epsilon_p, \epsilon_t)$

6 **return** $\{bit_i\}_{0\le i\le n}$

Table 2. Size of inputs of the `A2B` and `BooleanAllBitsOneTest` functions situated in Algorithm 5 for Scabbard's schemes and Saber

Function	Input Bytes			
	Florete	Sable	Espada	Saber
`A2B`	1824	1344	1544	1568
`BooleanAllBitsOneTest`	1248	1024	1304	1088

`BooleanAllBitsOneTest` for Florete are greater than Saber. So, the masked ciphertext comparison component of Florete needs more cycles than Saber. The masked input size of the function `A2B` of Espada is less than Saber, but the input size of `BooleanAllBitsOneTest` for Espada is bigger than Saber. So, the first-order masked comparison component is faster for Espada compared to Saber, but the second and third-order masked comparison component is slower in Espada than in Saber. However, the performance of each scheme's masked ciphertext comparison component in the suite Scabbard is better than Kyber because of the prepossessing steps needed in Kyber [14].

4 Performance Evaluation

We implemented all our algorithms on a 32-bit ARM Cortex-M4 microcontroller, STM32F407-DISCOVERY development board. We used the popular post-quantum cryptographic library and benchmarking framework PQM4 [21] for all measurements. The system we used to measure the performance of the masked implementations includes the compiler `arm-none-eabi-gcc` version `9.2.1`. The PQM4 library uses the system clock to measure the clock cycle, and

the frequency of this clock is 24 MHz. We employ random numbers to ensure the independence of the shares of the masked variable in masking algorithms. For this purpose, we use the on-chip `TRNG` (true random number generator) of the ARM Cortex-M4 device. This `TRNG` has a different clock frequency than the main system clock, which is 48 MHz. It generates a 32-bit random number in 40 clock cycles, equivalent to 20 clock cycles for the main system clock. Our implementations can be used for any order of masking. In this section, we provide the performance details of first-, second-, and third-order masking.

Table 3. Performance of $\texttt{MaskCBDSampler}_A$ and $\texttt{MaskCBDSampler}_B$

Order	×1000 clock cycles		
	1st	2nd	3rd
$\texttt{MaskCBDSampler}_A$	**178,591**	504,101	1,226,224
$\texttt{MaskCBDSampler}_B$	182,714	**499,732**	**909,452**

4.1 Analyzing the Performance of Masked CBD Samplers

As discussed in Sect. 3.5, $\texttt{MaskCBDSampler}_A$ and $\texttt{MaskCBDSampler}_B$ can be used for both Florete and Sable. Performance comparisons between $\texttt{MaskCBDSampler}_A$ and $\texttt{MaskCBDSampler}_B$ for different shares are provided in Table 3. Overall, we observe from the table that $\texttt{MaskCBDSampler}_B$ performs better than $\texttt{MaskCBDSampler}_A$ for higher-order masking. As a result, we use $\texttt{MaskCBDSampler}_B$ in the masked implementations of Florete and Sable.

4.2 Performance Measurement of Masked Scabbard Suite

Tables 4, 5, and 6 provide the clock cycles required to execute the masked decapsulation algorithm of Florete, Espada, and Sable, respectively. The overhead factors for the first-, second-, and third-order masked decapsulation operation of Florete are 2.74×, 5.07×, and 7.75× compared to the unmasked version. For Espada, the overhead factors for the first-, second-, and third-order decapsulation algorithm compared to the unmasked decapsulation are 1.78×, 2.82×, and 4.07, respectively. Similarly, for Sable, the overhead factors for the first-, second-, and third-order decapsulation algorithm are 2.38×, 4.26×, and 6.35× than the unmasked one. As mentioned earlier, the masked algorithm needs fresh random numbers to maintain security. Generating random numbers is a costly procedure. So, for a better understanding of the improvements, we also present the requirement of random bytes for Florete, Espada, and Sable in Table 7.

Table 4. Performance of Florete

Order	×1000 clock cycles							
	Unmask	1st		2nd		3rd		
Florete CCA-KEM-Decapsulation	954	2,621	(2.74×)	4,844	(5.07×)	7,395	(7.75×)	
CPA-PKE-Decryption	248	615	(2.47×)	1,107	(4.46×)	1,651	(6.65×)	
Polynomial arithmetic	241	461	(1.91×)	690	(2.86×)	917	(3.80×)	
Compression *original_msg*	6	153	(25.50×)	416	(69.33×)	734	(122.33×)	
Hash $\mathcal{G}$ (SHA3-512)	13	123	(9.46×)	242	(18.61×)	379	(29.15×)	
CPA-PKE-Encryption	554	1,744	(3.14×)	3,354	(6.05×)	5,225	(9.43×)	
Secret generation	29	427	(14.72×)	982	(33.86×)	1,663	(57.34×)	
XOF (SHAKE-128)	25	245	(9.80×)	484	(19.36×)	756	(30.24×)	
CBD (β_2)	4	182	(45.50×)	497	(124.25×)	907	(226.75×)	
Polynomial arithmetic *arrange_msg*	524	943	(2.51×)	1,357	(4.52×)	1,783	(6.79×)	
Polynomial Comparison		373		1,014		1,778		
Other operations	138	139	(1.00×)	140	(1.01×)	140	(1.01×)	

Table 5. Performance of Espada

Order	×1000 clock cycles							
	Unmask	1st		2nd		3rd		
Espada CCA-KEM-Decapsulation	2,422	4,335	(1.78×)	6,838	(2.82×)	9,861	(4.07×)	
CPA-PKE-Decryption	70	137	(1.95×)	230	(3.28×)	324	(4.62×)	
Polynomial arithmetic	69	116	(1.68×)	170	(2.46×)	225	(3.26×)	
Compression *original_msg*	0.4	20	(50.00×)	60	(150.00×)	99	(247.50×)	
Hash $\mathcal{G}$ (SHA3-512)	13	123	(9.46×)	243	(18.69×)	379	(29.15×)	
CPA-PKE-Encryption	2,215	3,950	(1.78×)	6,240	(2.81×)	9,031	(4.07×)	
Secret generation	57	748	(13.12×)	1,650	(28.94×)	3,009	(52.78×)	
XOF (SHAKE-128)	51	489	(9.58×)	968	(18.98×)	1,510	(29.60×)	
CBD (β_6)	6	259	(43.16×)	681	(113.50×)	1,498	(249.66×)	
Polynomial arithmetic *arrange_msg*	2,157	2,865	(1.44×)	3,593	(2.12×)	4,354	(2.79×)	
Polynomial Comparison		259		996		1,667		
Other operations	124	124	(1.00×)	124	(1.00×)	126	(1.01×)	

Table 6. Performance of Sable

Order	×1000 clock cycles							
	Unmask	1st		2nd		3rd		
Sable CCA-KEM-Decapsulation	1,020	2,431	(2.38×)	4,348	(4.26×)	6,480	(6.35×)	
CPA-PKE-Decryption	130	291	(2.23×)	510	(3.92×)	745	(5.73×)	
Polynomial arithmetic	128	238	(1.85×)	350	(2.73×)	465	(3.63×)	
Compression *original_msg*	2	52	(26.00×)	160	(80.00×)	280	(140.00×)	
Hash $\mathcal{G}$ (SHA3-512)	13	123	(9.46×)	242	(18.61×)	379	(29.15×)	
CPA-PKE-Encryption	764	1,903	(2.49×)	3,482	(4.55×)	5,241	(6.85×)	
Secret generation	29	427	(14.72×)	984	(33.93×)	1,666	(57.44×)	
XOF (SHAKE-128)	25	245	(9.80×)	484	(19.36×)	756	(30.24×)	
CBD (β_2)	4	182	(45.50×)	499	(124.75×)	909	(227.25×)	
Polynomial arithmetic *arrange_msg*	734	1,187	(2.00×)	1,640	(3.40×)	2,086	(4.86×)	
Polynomial Comparison		287		856		1,488		
Other operations	112	113	(1.00×)	113	(1.00×)	113	(1.00×)	

Table 7. Random number requirement for all the masked schemes of Scabbard

	# Random bytes								
	Florete			**Espada**			**Sable**		
Order	1st	2nd	3rd	1st	2nd	3rd	1st	2nd	3rd
CCA-KEM-Decapsulation	15,824	52,176	101,280	11,496	39,320	85,296	12,496	39,152	75,232
CPA-PKE-Decryption	2,560	10,176	20,352	304	1,216	2,432	832	3,328	6,656
Polynomial arithmetic	0	0	0	0	0	0	0	0	0
Compression	2,496	9,984	19,968	304	1,216	2,432	832	3,328	1,152
original_msg	64	192	384	0	0	0	0	0	0
Hash $\mathcal{G}$ (SHA3-512)	192	576	1,152	192	576	1,152	192	576	67,424
CPA-PKE-Encryption	13,072	41,424	79,776	11,000	37,528	81,712	11,472	35,248	6,656
Secret generation	6,528	16,512	29,952	4,896	14,688	35,520	6,528	16,512	29,952
XOF (SHAKE-128)	384	1,152	2,304	768	2,304	4,608	384	1,152	2,304
CBD (Binomial Sampler)	6,144	15,360	27,648	4,128	12,384	30,912	6,144	15,360	27,648
Polynomial arithmetic	0	0	0	0	0	0	0	0	0
arrange_msg	0	0	0	256	768	2,048	0	0	0
Polynomial Comparison	6,544	24,912	49,824	5,848	22,072	44,144	4,944	18,736	37,472
Other operations	0	0	0	0	0	0	0	0	0

4.3 Performance Comparison of Masked Scabbard Suite with the State-of-the-Art

We analyze the performance and random number requirements for masked decapsulation algorithms of Scabbard's schemes in comparison to the state-of-the-art masked implementations of LBC. We compare our masked Scabbard implementation with Bronchain et al.'s [10] and Bos et al.'s [9] masked implementations of Kyber and Kundu et al.'s [23] masked implementations of Saber in Table 8.

Table 8. Performance comparison of masked Scabbard implementations with the state-of-the-art

Scheme	Performance			# Randm numbers		
	(×1000 clock cycles)			(bytes)		
	1st	2nd	3rd	1st	2nd	3rd
Florete (this work)	2,621	4,844	7,395	15,824	52,176	101,280
Espada (this work)	4,335	6,838	9,861	11,496	39,320	85,296
Sable (this work)	**2,431**	**4,348**	**6,480**	12,496	39,152	75,232
Saber [23]	3,022	5,567	8,649	12,752	43,760	93,664
uSaber [23]	2,473	4,452	6,947	10,544	36,848	79,840
Kyber [10]	10,018	16,747	24,709	–	–	–
Kyber [9]	3,116*	44,347	115,481	12,072*	902,126	2,434,170

*: optimized specially for the first-order masking

First-, second- and third-order masked decapsulation implementations of Florete are respectively 73%, 71%, and 70% faster than Bronchain et al.'s [10] masked implementation of Kyber. Bos et al. optimized their algorithm specifically for the first-order masking of Kyber. Even though it is 15% slower than the

first-order masked decapsulation of Florete. Bos et al.'s [9] second- and third-order masked implementations of Kyber are respectively 89% and 93% slower than Florete. The random byte requirements in the masked version of Florete compared to Kyber are 94% less for the second order and 95% less for the third order. Florete also performs better than Saber. Florete needs 13%, 12%, and 14% fewer clock cycles than Saber for first-, second-, and third-order masking.

Masked decapsulation implementation of Espada performs 56%, 59%, and 60% better than Bronchain et al.'s [10] masked implementation of Kyber for first-, second-, and third-order, respectively. Second-, and third-order masked implementations of Espada are faster than Bos et al.'s [9] masked Kyber by 84% and 91%, respectively. The random bytes requirements in Espada compared to Kyber are 95% less for the second-order and 96% less for the third-order masking. Espada also uses fewer random numbers than Saber. Espada requires 9% fewer random bytes in first-order masking, 10% fewer random bytes in second-order masking, and 8% fewer random bytes in third-order masking than Saber.

We show that the masked implementation of Sable performs better than masked Kyber and Saber for first-, second-, and third-order (like Florete). Sable performs 75%, 74%, and 73% better than Bronchain et al.'s [10] masked implementation of Kyber and 21%, 90%, and 94% better than Bos et al.'s [9] masked implementation of Kyber first-, second-, and third-order, respectively. Compared to Kyber, Sable requires 95% and 96% less random bytes for second- and third-order masking. The performance of masked Sable is better than masked Saber by 19% for first-order, 21% for second-order, and 25% for third-order masking. Masked Sable uses 2%, 10%, and 19% less number of random bytes for first-, second-, and third-order than masked Saber, respectively. uSaber is a masking-friendly variant of Saber proposed during the third round of NIST submission. We notice that masked Sable is also faster than masked uSaber for arbitrary order. Masked Sable is 1% faster for first-order, 2% for second-order, and 6% for third-order than masked uSaber. Although first- and second-order masked Sable needs more random bytes than uSaber, third-order masked Sable requires 5% less random bytes than uSaber.

Implementations of masked Scabbard schemes achieve better performance and use fewer random bytes than masked Kyber because the schemes of Scabbard use the RLWR/ MLWR problem as an underlying hard problem and Kyber uses the MLWE problem as the hard problem. The decapsulation operation of RLWR/ MLWR-based KEM has fewer components compared to the decapsulation operation of RLWE/ MLWE-based KEM due to the requirement of sampling error vectors and polynomials generations in the re-encryption step of RLWE/ MLWE-based KEMs. RLWR/ MLWR-based KEMs also benefit due to the use of power-of-two moduli. Computationally expensive components, such as A2B or B2A conversions, are cheaper when using power-of-two moduli. The schemes of Scabbard also use slightly smaller parameters than Kyber, which also contributes to achieving better performance and requirements of fewer random bytes for masked implementation of Scabbard's KEMs compared to Kyber.

5 Conclusions

In this work, we presented the impact of different design decisions of LBC on masking. We analyzed each component where masking is needed and discussed each design decision's positive and negative impact on performance. As we mentioned at the beginning of the paper, it is possible to improve different practical aspects, such as masking overheads, by modifying the existing designs of PQC. This highlights the necessity of further research efforts to improve existing PQC designs.

Acknowledgements. This work was partially supported by Horizon 2020 ERC Advanced Grant (101020005 Belfort), CyberSecurity Research Flanders with reference number VR20192203, BE QCI: Belgian-QCI (3E230370) (see beqci.eu), and Intel Corporation. Angshuman Karmakar is funded by FWO (Research Foundation - Flanders) as a junior post-doctoral fellow (contract number 203056/1241722N LV).

References

1. Alagic, G., et al.: Status report on the third round of the NIST post-quantum cryptography standardization process (2022). Accessed 26 June 2023
2. Alkim, E., Ducas, L., Pöppelmann, T., Schwabe, P.: Post-quantum key exchange - a new hope. In: Holz, T., Savage, S. (eds.) 25th USENIX Security Symposium, USENIX Security 2016, Austin, TX, USA, 10–12 August 2016, pp. 327–343. USENIX Association (2016)
3. Banerjee, A., Peikert, C., Rosen, A.: Pseudorandom functions and lattices. In: Pointcheval, D., Johansson, T. (eds.) EUROCRYPT 2012. LNCS, vol. 7237, pp. 719–737. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-29011-4_42
4. 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)
5. Bermudo Mera, J.M., Karmakar, A., Kundu, S., Verbauwhede, I.: Scabbard: a suite of efficient learning with rounding key-encapsulation mechanisms. IACR Trans. Cryptogr. Hardw. Embed. Syst. **2021**(4), 474–509 (2021)
6. 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
7. 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)
8. 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. Cryptology ePrint Archive, Report 2017/634 (2017)
9. 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)
10. 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)

11. 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
12. Cheon, J.H., Choe, H., Hong, D., Yi, M.: SMAUG: pushing lattice-based key encapsulation mechanisms to the limits. Cryptology ePrint Archive, Paper 2023/739 (2023)
13. Coron, J.-S.: Higher order masking of look-up tables. In: Nguyen, P.Q., Oswald, E. (eds.) EUROCRYPT 2014. LNCS, vol. 8441, pp. 441–458. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-642-55220-5_25
14. D'Anvers, J., Beirendonck, M.V., Verbauwhede, I.: Revisiting higher-order masked comparison for lattice-based cryptography: algorithms and bit-sliced implementations. IEEE Trans. Comput. **72**(2), 321–332 (2023)
15. 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
16. Espitau, T., et al.: Mitaka: a simpler, parallelizable, maskable variant of FALCON. In: Dunkelman, O., Dziembowski, S. (eds.) EUROCRYPT 2022. LNCS, vol. 13277, pp. 222–253. Springer, Cham (2022). https://doi.org/10.1007/978-3-031-07082-2_9
17. Fouque, P.A., et al.: Falcon: Fast-Fourier lattice-based compact signatures over NTRU (2018). Accessed 28 June 2023
18. Fujisaki, E., Okamoto, T.: How to enhance the security of public-key encryption at minimum cost. In: Imai, H., Zheng, Y. (eds.) PKC 1999. LNCS, vol. 1560, pp. 53–68. Springer, Heidelberg (1999). https://doi.org/10.1007/3-540-49162-7_5
19. Gross, H., Schaffenrath, D., Mangard, S.: Higher-order side-channel protected implementations of KECCAK. Cryptology ePrint Archive, Report 2017/395 (2017)
20. Jiang, H., Zhang, Z., Chen, L., Wang, H., Ma, Z.: Post-quantum IND-CCA-secure KEM without additional hash. IACR Cryptology ePrint Archive 2017/1096 (2017)
21. Kannwischer, M.J., Rijneveld, J., Schwabe, P., Stoffelen, K.: PQM4: post-quantum crypto library for the ARM Cortex-M4. https://github.com/mupq/pqm4
22. KpqC: Korean PQC competition (2022). https://www.kpqc.or.kr/competition.html. Accessed 30 June 2023
23. Kundu, S., D'Anvers, J.P., Van Beirendonck, M., Karmakar, A., Verbauwhede, I.: Higher-order masked saber. In: Galdi, C., Jarecki, S. (eds.) SCN 2022. LNCS, vol. 13409, pp. 93–116. Springer, Cham (2022). https://doi.org/10.1007/978-3-031-14791-3_5
24. Langlois, A., Stehlé, D.: Worst-case to average-case reductions for module lattices. Des. Codes Cryptogr. **75**(3), 565–599 (2015). https://doi.org/10.1007/s10623-014-9938-4
25. 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
26. Rivain, M., Prouff, E.: Provably secure higher-order masking of AES. In: Mangard, S., Standaert, F.-X. (eds.) CHES 2010. LNCS, vol. 6225, pp. 413–427. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-15031-9_28
27. 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)

28. 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
29. Shamir, A.: How to share a secret. Commun. ACM **22**(11), 612–613 (1979)
30. Yao, A.C.: Protocols for secure computations. In: 23rd Annual Symposium on Foundations of Computer Science (SFCS 1982), pp. 160–164 (1982)