

# High-order Polynomial Comparison and Masking Lattice-based Encryption

Jean-Sébastien Coron<sup>1</sup>, François Gérard<sup>1</sup>, Simon Montoya<sup>2,3</sup>, and Rina Zeitoun<sup>2</sup>

<sup>1</sup> University of Luxembourg, Esch-sur-Alzette, Luxembourg  
jean-sebastien.coron@uni.lu, francois.gerard@uni.lu

<sup>2</sup> IDEMIA, Cryptography & Security Labs, Courbevoie, France  
simon.montoya@idemia.com, rina.zeitoun@idemia.com

<sup>3</sup> LIX, INRIA, CNRS, École Polytechnique, Institut Polytechnique de Paris, France

**Abstract** The main protection against side-channel attacks consists in computing every function with multiple shares via the masking countermeasure. For IND-CCA secure lattice-based encryption schemes, the masking of the decryption algorithm requires the high-order computation of a polynomial comparison. In this paper, we describe and evaluate a number of different techniques for such high-order comparison, always with a security proof in the ISW probing model. As an application, we describe the full high-order masking of the NIST standard Kyber, with a concrete implementation on ARM Cortex M architecture, and a  $t$ -test evaluation.

## 1 Introduction

**Post-quantum cryptography.** The most widely used public-key cryptosystems today are based on RSA and ECC, but they are breakable in polynomial time using a quantum computer. While it is currently unknown whether building a scalable quantum computer is feasible or not, the goal of post-quantum cryptography is to design alternatives to RSA and ECC with resistance against quantum attacks. Initiated in 2016, the NIST post-quantum standardization has now entered its final round, with the selection of the Kyber algorithm [BDK<sup>+</sup>18, ABD<sup>+</sup>21] for general encryption. The security of Kyber is based on the hardness of the module learning-with-errors (M-LWE) problem, which is conjectured to remain hard even with a full-scale quantum computer.

**Side-channel attacks and the masking countermeasure.** Lattice-based public-key encryption schemes are vulnerable to side-channel attacks as any other cryptosystems, see for example [PPM17, HCY20, XPRO20]. The main countermeasure against side-channel attacks is masking [CJRR99]. It consists in splitting every variable  $x$  into  $n$  shares with  $x = x_1 + \dots + x_n$ , and processing the shares separately. Then, an adversary with a limited number of probes cannot learn more than an adversary without probes. The study of protecting circuits against high-order attacks was initiated by Ishai, Sahai and Wagner in [ISW03]. They considered an adversary who can probe at most  $t$  wires in a circuit. They showed how to transform any Boolean circuit  $C$  into a circuit of size  $\mathcal{O}(|C| \cdot t^2)$  secure against such adversary, using  $n = 2t + 1$  shares. This was later improved by Barthe *et al.* to  $n = t + 1$  shares only [BBD<sup>+</sup>16], who introduced the notions of (Strong) Non-Interference (NI/SNI) to facilitate the writing of security proofs with the composition of gadgets.

The masking countermeasure was initially developed for securing block-ciphers against side-channel attacks, for example AES in [RP10]. It appears that securing lattice-based schemes against high-order attacks offers quite new and interesting challenges, thanks to the rich algorithmic diversity of post-quantum cryptography. While in principle any algorithm can be written

as a Boolean circuit  $C$  and then secured by applying [ISW03] with complexity  $\mathcal{O}(|C| \cdot n^2)$  for  $n$  shares, very often that would be too inefficient. For example, lattice-based schemes usually combine Boolean and arithmetic operations, so for efficiency reasons one must repeatedly convert between arithmetic and Boolean masking. Such high-order conversions were previously considered modulo  $2^k$  in [CGV14], with complexity  $\mathcal{O}(n^2 \cdot k)$  in both directions.<sup>1</sup> This has been recently extended to prime moduli in [BBE<sup>+</sup>18] for the high-order masking of the GLP lattice-based signature scheme. Therefore post-quantum cryptography is an opportunity to enrich the tool set of high-order masking.

**High-order masking of lattice-based schemes.** In this paper, we consider the high-order masking of the IND-CCA decryption of lattice-based schemes, and more specifically the Kyber scheme selected by NIST for standardization. The Fujisaki-Okamoto (FO) transformation [FO99] starts from an IND-CPA secure public-key encryption scheme, and transforms it into an IND-CCA secure PKE generically. Informally, for IND-CCA encryption, a hash of the message  $m$  is used to generate the random coins in the basic IND-CPA encrypt procedure. During IND-CCA decryption, this is verified via re-encryption. More precisely, the IND-CCA decryption of lattice-based schemes such as Kyber and Saber comprises the following steps:

1. IND-CPA decryption of the ciphertext  $c$  to obtain a message  $m$
2. Re-encryption of  $m$  into a ciphertext  $c'$ ; this includes the binomial sampling of the error polynomials computed from the hash of  $m$
3. Polynomial comparison between  $c$  and  $c'$ .

In the following, we recall the state of the art for high-order masking those three steps. In [SPOG19], the authors described an efficient technique for high-order masking the binomial sampling in the re-encryption of  $m$  at Step 2 above, based on a 1-bit Boolean to arithmetic conversion modulo  $q$  with complexity  $\mathcal{O}(n^2)$ ; their technique is an extension of a first-order algorithm from [OSPG18]. In [CGMZ22], the authors described high-order table-based conversion algorithms between arithmetic and Boolean masking, based on the randomized table countermeasure from [Cor14]. For high-order masking the binomial sampling in the re-encryption of Kyber and Saber (Step 2), this provides a slightly simpler alternative to [SPOG19], but with the same  $\mathcal{O}(n^2)$  complexity. For the high-order masking of IND-CPA decryption (Step 1), the authors also described a modulus switching technique combined with a fast table-based arithmetic to Boolean conversion, that offers significant efficiency improvement compared to previous techniques.

Note that to obtain a fully masked implementation, all three steps must be masked, otherwise this can lead to a CCA attack. For example, if the plaintext  $m$  is not masked at Step 2, an attacker could submit closely related ciphertexts  $c'$  and detect when it decrypts into some  $m' \neq m$ , which would reveal information about the secret key. Similarly, the polynomial comparison at Step 3 should be correctly masked, otherwise this can also lead to a CCA attack. For example, in [BDH<sup>+</sup>21] the authors show that the first-order ciphertext comparison from [OSPG18] is insecure, because the comparison is performed iteratively on different parts of the ciphertext: the result of the first comparison leaks information to the attacker and leads to a CCA attack. The authors of [BDH<sup>+</sup>21] also considered a similar attack against the high-order polynomial comparison from [BPO<sup>+</sup>20], which processed successive blocks of coefficients, and the pass/fail bit was computed in the clear for each block, which also led to a CCA attack.

---

<sup>1</sup> This complexity can be improved to  $\mathcal{O}(n^2 \cdot \log k)$  using [CGTV15], however for usual modulus size (say  $k \leq 32$ ) the number of operations is similar, so in this paper for simplicity we use the  $\mathcal{O}(n^2 \cdot k)$  complexity.

**High-order polynomial comparison.** The above attacks show that the polynomial comparison (Step 3) must be an atomic operation that does not leak partial comparison results on a subset of the coefficients. This is actually required to get any hope of a security proof: namely in the simulation-based approach, the simulator only gets the final bit  $b$  of the ciphertext comparison (since the attacker eventually learns the result of the comparison, this bit  $b$  can be given for free to the simulator), but cannot possibly simulate intermediate pass/fail bits, since that would require the knowledge of the secret key.

When the coefficients of the ciphertexts are Boolean masked, ciphertext comparison is relatively straightforward via a high-order Boolean circuit. Firstly, for each coefficient in  $\{0, 1\}^k$ , one performs a xor between the two ciphertexts (over the Boolean shares), secondly a high-order And of the complement in  $\{0, 1\}^k$ , followed by a final zero-testing of the result to get a single bit. The complexity is  $\mathcal{O}(\ell n^2 + n^2 \log k)$  for  $\ell$  coefficients and  $n$  shares, since one must perform  $\ell$  secure And's, each of complexity  $\mathcal{O}(n^2)$ , followed by a final zero-test of complexity  $\mathcal{O}(n^2 \log k)$ . We recall this procedure in Appendix D.1, with the `PolyZeroTestBool` algorithm.

Comparing two coefficients  $x, y$  in some additive group  $G$  is equivalent to zero-testing  $x - y \in G$ , so in the following we focus on zero-testing. When the coefficients of the ciphertexts are masked modulo  $2^k$  or a  $k$ -bit prime  $q$ , a straightforward approach is to first perform an arithmetic to Boolean conversion of each coefficient, so that the zero-testing can be performed over Boolean shares as previously. We describe the corresponding `PolyZeroTestAB` algorithm in Appendix D.2. The complexity is then  $\mathcal{O}(\ell n^2 k)$ , because for  $\ell$  coefficients one must perform  $\ell$  arithmetic to Boolean conversions, each with complexity  $\mathcal{O}(n^2 k)$ ; see Table 1 for a summary.

When working with arithmetic shares modulo a prime  $q$  as in Kyber, it is actually advantageous to first reduce the zero-testing of the  $\ell$  coefficients to the zero-testing of  $\kappa \ll \ell$  coefficients [BDH<sup>+</sup>21]. This is done via computing  $\kappa$  linear combinations over all coefficients of the polynomial.<sup>2</sup> If at least one coefficient is non-zero, such linear combination will be non-zero, except with probability  $1/q$ . For  $\kappa$  linear combinations, the error probability decreases exponentially to  $q^{-\kappa}$ . To get an error probability lower than  $2^{-\lambda}$ , one can therefore take  $\kappa = \lceil \lambda / \log_2 q \rceil$  linear combinations, where  $\lambda$  is the security parameter. This approach is quite efficient as it has complexity  $\mathcal{O}(\ell \kappa \cdot n)$  only, which is linear in  $n$  (as opposed to quadratic for most high-order gadgets). This is because the coefficients of the linear sums need not be masked, and therefore each multiplication has linear complexity in  $n$ . We recall the algorithm in Appendix D.3. The authors of [BDH<sup>+</sup>21] only described the reduction of zero-testing from  $\ell$  coefficients to  $\kappa \ll \ell$  coefficients, and did not describe a full zero-testing procedure. To obtain a full zero-testing procedure, one can use the `PolyZeroTestAB` algorithm above for the zero-testing of the remaining  $\kappa$  coefficients.

In [BGR<sup>+</sup>21], the authors described the first completely masked implementation of Kyber, secure against first-order and higher-order attacks. For the IND-CPA decryption (Step 1), the authors consider the threshold function  $\text{th}(x)$  outputting 0 if  $x < q/2$  and 1 otherwise. They show that  $\text{th}(x) = x_{11} \oplus (\neg x_{11} \cdot x_{10} \cdot x_9 \cdot (x_8 \oplus (\neg x_8 \cdot x_7)))$ , where  $x_i$  is the  $i$ -th bit of  $x$ ; namely this corresponds to a binary comparison with the threshold  $\lfloor q/2 \rfloor$ . This implies that the high-order computation of  $\text{th}(x)$  can be performed by first converting the masking of  $x$  from arithmetic modulo  $q$  to Boolean, using [BBE<sup>+</sup>18]; then  $\text{th}(x)$  can be computed with high-order secure implementations of the And and Xor gadgets. For the high-order polynomial comparison (Step 3), the technique of [BGR<sup>+</sup>21] consists in performing the comparison with uncompressed ciphertexts. The advantage of this approach is that the ciphertext compression from Kyber does not need to be explicitly masked. Given the masked uncompressed polynomials obtained from

<sup>2</sup> instead of partial subsets of the coefficients as in [BPO<sup>+</sup>20], which lead to a CCA attack [BDH<sup>+</sup>21]

re-encryption, the ciphertext comparison requires checking that every coefficient belongs to a certain public range modulo  $q$ . Such a range test is performed via two high-order comparisons, based on arithmetic to Boolean conversions.

Recently, a high-order algorithm for performing the comparison between two ciphertexts was described in [DHP<sup>+</sup>22], based on computing linear sums as in [BDH<sup>+</sup>21], but by switching to a larger modulus, so that the probability of an incorrect result with a single equality check becomes negligible. As opposed to [BDH<sup>+</sup>21], the technique works for both prime and power-of-two moduli  $q$ . More precisely, it converts into a larger modulus  $2^{\alpha_p + \lambda}$  where  $\lambda$  is the security parameter, so that the false positive probability is at most  $2^{-\lambda}$ , with  $\alpha_p = 10$  for Kyber and Saber. The final zero-testing is performed using arithmetic to Boolean conversion. The complexity is dominated by the cost of the conversion of each coefficient into the larger modulus, which comprises an arithmetic to Boolean conversion (with complexity  $\mathcal{O}(n^2 \cdot k)$  where  $k = \lceil \log_2 q \rceil$  is the modulus size), and a Boolean to arithmetic conversion to the larger modulus (with complexity  $\mathcal{O}(n^2 \cdot \lambda)$  for security parameter  $\lambda$ ). Asymptotically, the final complexity is then  $\mathcal{O}(\ell \cdot n^2 (k + \lambda))$  for  $\ell$  coefficients, which is similar to the PolyZeroTestAB approach recalled above (see Table 1 below for a comparison). The authors also described an efficient implementation of their technique, faster in practice than [BGR<sup>+</sup>21], and also a verification of the side-channel security of their implementation using concrete leakages.

**Our contributions.** Our first contribution is to describe two new techniques for performing the high-order ciphertext comparison for a prime modulus  $q$ , more efficient than with arithmetic to Boolean conversion. Our first technique is based on converting from arithmetic masking modulo  $q$  to multiplicative masking, which enables us to perform a zero-test of  $x$  without revealing more information about  $x$ . More precisely, starting from the arithmetic shares  $x_i$  of  $x = x_1 + \dots + x_n \pmod{q}$ , we first convert into a multiplicative sharing  $u_1 \cdot \dots \cdot u_n \cdot x = B \pmod{q}$ . With invertible masks  $u_i \in \mathbb{Z}_q^*$ , we must have  $B \neq 0$  if  $x \neq 0$ , and  $B = 0$  if  $x = 0$ , which gives a zero-test of  $x$ . We prove that an adversary with at most  $n - 1$  probes does not learn more information about  $x$ . For a single coefficient, the complexity is  $\mathcal{O}(n^2)$ , instead of  $\mathcal{O}(n^2 \cdot k)$  with previous approaches. For zero-testing  $\ell$  coefficients, we first apply the technique from [BDH<sup>+</sup>21] to reduce to the zero-testing of  $\kappa \ll \ell$  coefficients, for  $\kappa = \lceil \lambda / \log_2 q \rceil$ . For zero-testing the remaining  $\kappa$  coefficients all at once, we again use  $\kappa$  linear combinations, but this time with masked coefficients, so that we can eventually zero-test each linear combination separately, using the above method. We describe the corresponding PolyZeroTestMult algorithm in Section 4.1.

Our second algorithm is based on masked exponentiation modulo a prime  $q$ , using Fermat’s little theorem. For zero-testing a single arithmetically masked coefficient  $x$ , we high-order compute  $b = 1 - x^{q-1} \pmod{q}$ , which gives  $b = 1$  if  $x = 0$ , and  $b = 0$  if  $x \neq 0$ , as required. With a square-and-multiply, the complexity is  $\mathcal{O}(n^2 \cdot \log q)$ . For zero-testing  $\ell$  coefficients, as previously we first reduce to the zero-testing of  $\kappa \ll \ell$  coefficients  $y^{(j)}$ . For zero-testing the remaining  $\kappa$  coefficients  $y^{(j)}$  all at once, we high-order compute the product modulo  $q$  of the corresponding bits  $b^{(j)} = 1 - (y^{(j)})^{q-1} \pmod{q}$ , which gives  $b = 1$  if all coefficients are zero, and  $b = 0$  otherwise, as required. We describe the corresponding PolyZeroTestExpo algorithm in Section 4.2. We refer to Table 1 below for a summary. We show that in practice, both techniques are more efficient than arithmetic to Boolean conversion (see Section 3.3).

As an application, our second contribution is to improve the efficiency of the high-order polynomial comparison in Kyber for IND-CCA decryption (Step 3). Recall that in Kyber the ciphertext coefficients are compressed from modulo  $q$  to  $d$  bits by computing the function

	Technique	Multi coef.	Masking	Complexity
PolyZeroTestBool	Secure And	SecAnd	Boolean	$\mathcal{O}(\ell n^2 + n^2 \log k)$
PolyZeroTestAB	A $\rightarrow$ B conv.	SecAnd	mod $q, 2^k$	$\mathcal{O}(\ell n^2 k)$
[DHP <sup>+</sup> 22]	A $\rightarrow$ B conv.	Linear comb.	mod $q, 2^k$	$\mathcal{O}(\ell n^2 (k + \lambda))$
PolyZeroTestMult	Mult. masking	Linear comb.	mod $q$	$\mathcal{O}(\ell \kappa n + \kappa^2 n^2)$
PolyZeroTestExpo	Exponentiation	SecMult	mod $q$	$\mathcal{O}(\ell \kappa n + \kappa n^2 \log q)$

Table 1: Complexities of polynomial comparison, with  $\ell$  coefficients and  $n$  shares, and a modulus  $2^k$  or a  $k$ -bit prime  $q$ . We write  $\kappa = \lceil \lambda / \log_2 q \rceil$ , where  $\lambda$  is the security parameter.

$\text{Compress}_{q,d}(x) := \lfloor (2^d/q) \cdot x \rfloor \bmod 2^d$ . We first consider an alternative approach to [BGR<sup>+</sup>21], where we explicitly high-order mask the `Compress` function during the encryption process. For this we extend the modulus switching technique from [FBR<sup>+</sup>21] which was first-order only. To our knowledge, this is the first proposal for high-order masking the `Compress` function of Kyber<sup>3</sup>. We also consider the high-order ciphertext comparison without the `Compress` function as in [BGR<sup>+</sup>21], and we provide an alternative, faster technique when the output size of `Compress` is close to the bitsize of  $q$ , which is the case for 3/4 of the ciphertext coefficients in Kyber. Finally, we show that the best strategy for polynomial comparison in Kyber is hybrid: for the first part of the ciphertext, we do not apply the `Compress` function and perform the comparison over uncompressed ciphertexts (as in [BGR<sup>+</sup>21], but with our faster algorithm), while for the second part of the ciphertext, we high-order compute the `Compress` function and perform the comparison over Boolean shares.

Finally, we provide a detailed description of the masking of the full IND-CCA decryption of the Kyber scheme at any order. We also describe the practical results of a C implementation of the full high-order masking of Kyber and Saber. The source code is public and can be found at

[https://github.com/fragerar/HOTableConv/tree/main/Masked\\_KEMs](https://github.com/fragerar/HOTableConv/tree/main/Masked_KEMs)

Finally, we have also performed a  $t$ -test evaluation using the ChipWhisperer platform for power traces.

**Follow-up works.** Very recently, [DBV22] described a variant of our hybrid method for the ciphertext comparison in Kyber, in which for the first part of the ciphertext, the results of the comparison over uncompressed coefficients are converted from arithmetic to Boolean masking. While the asymptotic complexity remains the same, for a bitsliced implementation, the authors obtain a 25% speed-up factor compared to our hybrid method. Very recently, the authors of [BC22] also described a bitsliced implementation of Kyber and Saber, starting from our implementation of the hybrid method, and achieved significant performance improvement.

## 2 Notations and security definitions

For any positive integer  $q$ , we define  $r' = r \bmod q$  to be the unique element  $r'$  in the range  $[0, q[$  such that  $r' = r \pmod{q}$ . For an even (resp. odd) positive integer  $q$ , we define  $r' = r \bmod^{\pm} q$  to

<sup>3</sup> In [BGR<sup>+</sup>21] and [CGMZ22], the masking of the `Compress` function was considered only for  $d = 1$  bit as output, as used in IND-CPA decryption.

be the unique element  $r'$  in the range  $-q/2 < r' \leq q/2$  (resp.  $-(q-1)/2 \leq r' \leq (q-1)/2$ ) such that  $r' = r \pmod{q}$ . For  $x \in \mathbb{Q}$ , we denote by  $\lfloor x \rfloor$  the rounding of  $x$  to the nearest integer, with ties being rounded up. We denote by  $x \gg k$  the shifting of an integer  $x$  with  $k$  positions to the right, that is  $\lfloor x/2^k \rfloor$ .

We recall below the NI/SNI definitions introduced in [BBD<sup>+</sup>16]. Those definitions are quite convenient as they allow the easy composition of gadgets. One can then focus on proving the NI/SNI property for individual gadgets, and the security of the full circuit will follow by composition. The SNI definition is stronger than NI in that the number of input shares required for the simulation only depends on the number of internal probes, and not on the number of output shares that must be simulated. If a gadget only satisfies the NI definition, usually this is not a problem as we can apply some SNI mask refreshing as output and the resulting gadget becomes SNI (see [BBD<sup>+</sup>16]). In this paper all our gadgets will be proven either NI or SNI.

**Definition 1 (*t*-NI security).** *Let  $G$  be a gadget taking as input  $(x_i)_{1 \leq i \leq n}$  and outputting the vector  $(y_i)_{1 \leq i \leq n}$ . The gadget  $G$  is said *t*-NI secure if for any set of  $t_1 \leq t$  intermediate variables, there exists a subset  $I$  of input indices with  $|I| \leq t_1$ , such that the  $t_1$  intermediate variables can be perfectly simulated from  $x_{|I}$ .*

**Definition 2 (*t*-SNI security).** *Let  $G$  be a gadget taking as input  $n$  shares  $(x_i)_{1 \leq i \leq n}$ , and outputting  $n$  shares  $(z_i)_{1 \leq i \leq n}$ . The gadget  $G$  is said to be *t*-SNI secure if for any set of  $t_1$  probed intermediate variables and any subset  $\mathcal{O}$  of output indices, such that  $t_1 + |\mathcal{O}| \leq t$ , there exists a subset  $I$  of input indices that satisfies  $|I| \leq t_1$ , such that the  $t_1$  intermediate variables and the output variables  $z_{|\mathcal{O}}$  can be perfectly simulated from  $x_{|I}$ .*

Note that for masking the IND-CCA decryption, when performing the comparison between two ciphertexts  $c$  and  $c'$ , the output bit  $b$  of the comparison must eventually be computed in the clear, which means that the  $n$  shares  $b_i$  of  $b$  must eventually be recombined. For this we use the extended notion of NI security from [BBE<sup>+</sup>18, Definition 7], in which the output  $b$  of the gadget is given to the simulator.

**Definition 3 (*t*-Nlo security [BBE<sup>+</sup>18]).** *Let  $G$  be a gadget taking as input  $(x_i)_{1 \leq i \leq n}$  and outputting  $b$ . The gadget  $G$  is said *t*-Nlo secure if for any set of  $t_1 \leq t$  intermediate variables, there exists a subset  $I$  of input indices with  $|I| \leq t_1$ , such that the  $t_1$  intermediate variables can be perfectly simulated from  $x_{|I}$  and  $b$ .*

To satisfy this definition, one can use the same approach as in [Cor14] for recombining the output shares of a block-cipher to output the ciphertext: one performs a sequence of  $n$  mask refreshing, each of complexity  $\mathcal{O}(n)$ , so that the share recombination can be perfectly simulated, knowing the output bit  $b$ . Namely, this output bit  $b$  of the comparison can be given for free to the simulator, since that bit  $b$  is eventually known by the adversary. Following [Cor14], this enables to prove the  $(n-1)$ -Nlo property of the share recombination algorithm; see Appendix B.1 for more details.

In Appendix B.2, we describe a slightly more efficient approach, still with complexity  $\mathcal{O}(n^2)$ , but using only half the randomness.

### 3 High-order zero testing

In the IND-CCA decryption of lattice-based schemes such as Kyber, according to the Fujisaki-Okamoto transform, we must perform a comparison between the input ciphertext  $\tilde{c}$ , and the

re-encrypted ciphertext  $c$ . In the context of the masking countermeasure, the re-encrypted ciphertext  $c$  is masked with  $n$  shares, so we must perform this comparison over arithmetic or Boolean shares. Moreover, the coefficients of the polynomials  $\tilde{c}$  and  $c$  must be compared all at once. Otherwise the leaking of partial comparison results can leak information about the secret key, as demonstrated in [BDH<sup>+</sup>21].

In this section, for simplicity, we consider the zero-testing of a single coefficient. We will then show in Section 4 how to test multiple coefficients at once. With arithmetic shares, comparing two individual coefficients  $x$  and  $y$  in  $\mathbb{Z}_q$  is equivalent to zero testing  $x - y \in \mathbb{Z}_q$ . Similarly, with Boolean shares, comparing two coefficients  $x, y \in \{0, 1\}^k$  is equivalent to zero testing  $x \oplus y$ . Therefore, in the rest of this section, we focus on zero-testing.

For a single coefficient  $x$ , we are therefore given as input the  $n$  Boolean shares of  $x = x_1 \oplus \dots \oplus x_n \in \{0, 1\}^k$ , or the  $n$  arithmetic shares of  $x = x_1 + \dots + x_n \bmod q$ , and we must output a bit  $b$ , with  $b = 1$  if  $x = 0$  and  $b = 0$  if  $x \neq 0$ , without revealing more information about  $x$ . This means that an adversary with at most  $t = n - 1$  probes will learn nothing about  $x$ , except if  $x = 0$  or not. For the security proof, the simulation technique is the same as for security proofs in the ISW probing model, except that the output bit  $b$  is additionally given to the simulator (see Section 2).

From Boolean shares over  $\{0, 1\}^k$ , one can perform a zero-test with complexity  $\mathcal{O}(n^2 \cdot \log k)$ ; we recall the technique in Appendix C.3 (ZeroTestBoolLog algorithm). From arithmetic shares modulo  $q$ , the simplest technique is to first perform an arithmetic to Boolean conversion, and then apply the zero-testing on the Boolean shares. The complexity is  $\mathcal{O}(n^2 k)$  for a  $k$ -bit modulus. We recall the technique in Appendix C.5 (ZeroTestAB algorithm). In Appendix C.6, we also describe an alternative zero-testing for arithmetic masking based on the generic table recomputation approach from [CGMZ22], with the register optimization. In that case the countermeasure has complexity  $\mathcal{O}(n^2)$  only, assuming that we have access to  $2^k$ -bit registers. Therefore this optimization can only work for small  $k$ , say up to  $k = 8$ .

**New zero-testing gadgets.** For arithmetic shares modulo a prime  $q$ , we describe two new zero-testing algorithms, more efficient than the state of the art. The first technique (ZeroTestMult in Section 3.1) is based on converting from arithmetic masking to multiplicative masking, so that one can distinguish between  $x = 0$  and  $x \neq 0$ , without revealing more information about  $x$ . The second technique (ZeroTestExpo in Section 3.2) is based on Fermat’s theorem and consists in high-order computing  $b = 1 - x^{q-1} \bmod q$ , which gives  $b = 1$  if  $x = 0$ , and  $b = 0$  if  $x \neq 0$ , as required. We will see in Section 4 that for zero-testing  $\ell$  coefficients at once, these two techniques are more efficient than arithmetic to Boolean conversion. We refer to Table 2 for a summary.

	Technique	Masking	Complexity
ZeroTestBoolLog	Secure And	Boolean	$\mathcal{O}(n^2 \cdot \log k)$
ZeroTestAB	A $\rightarrow$ B conversion	mod $q, 2^k$	$\mathcal{O}(n^2 \cdot k)$
ZeroTestMult	Mult. masking	mod $q$	$\mathcal{O}(n^2)$
ZeroTestExpo	Exponentiation	mod $q$	$\mathcal{O}(n^2 \cdot \log q)$

Table 2: Complexities of zero testing a single value with  $n$  arithmetic shares, and a modulus  $2^k$  or a  $k$ -bit prime  $q$ .

For the ciphertext comparison in Kyber, we will describe in Section 5 a hybrid approach in which the first part of the re-encrypted ciphertext is arithmetically masked modulo  $q$ , while the remaining part is Boolean masked. Therefore, we will use the `ZeroTestBoolLog` algorithm for the second part, and for the first part either `ZeroTestMult` or `ZeroTestExpo`, which offer similar performances on Kyber. For Saber, the re-encrypted ciphertext is completely Boolean shared, so we will use `ZeroTestBoolLog`. Finally, the `ZeroTestAB` algorithm will not be used in our constructions, but we keep this algorithm anyway for comparison.

### 3.1 Zero testing modulo a prime $q$ via multiplicative masking

Our technique works for prime  $q$  only. It is based on converting from arithmetic masking modulo  $q$  to multiplicative masking. When the secret value  $x$  is 0, the multiplicatively masked value remains 0, whereas for  $x \neq 0$ , we obtain a random non-zero masked value. This enables us to distinguish the two cases, without leaking more information about  $x$ .

More precisely, given as input the shares  $x_i$  of  $x = x_1 + \dots + x_n \pmod{q}$ , we convert the arithmetic masking into a multiplicative masking. For this we generate a random  $u_1 \in \mathbb{Z}_q^*$  and we compute:

$$u_1 \cdot x = u_1 \cdot x_1 + \dots + u_1 \cdot x_n \pmod{q}$$

by computing the corresponding shares  $x'_i = u_1 \cdot x_i \pmod{q}$  for all  $1 \leq i \leq n$ . We then perform a linear mask refreshing of the arithmetic shares  $x'_i$ . Such linear mask refreshing is not SNI but it is NI. Moreover, its property is that any subset of  $n - 1$  output shares is uniformly and independently distributed, as in the mask refreshing from [RP10].

We proceed similarly with the multiplicative shares  $u_2, \dots, u_n \in \mathbb{Z}_q^*$ . Eventually we obtain an arithmetic sharing  $(B_i)_{1 \leq i \leq n}$  satisfying:

$$u_1 \cdot \dots \cdot u_n \cdot x = B_1 + \dots + B_n \pmod{q}$$

Thanks to the  $n$  multiplicative shares  $u_i$ , we can now safely decode the arithmetic sharing  $(B_i)_{1 \leq i \leq n}$  without revealing more information about  $x$ . More precisely, we compute  $B = B_1 + \dots + B_n \pmod{q}$ , and we obtain:

$$u_1 \cdot \dots \cdot u_n \cdot x = B \pmod{q}$$

Recall that  $u_i \in \mathbb{Z}_q^*$  for all  $1 \leq i \leq n$ . Therefore if  $x \neq 0$ , we must have  $B \neq 0$ , and if  $x = 0$ , we have  $B = 0$ . This gives a zero-test of  $x$ . We provide below a pseudocode description of the corresponding `ZeroTestMult` algorithm. We recall the `LinearRefreshMasks` algorithm in Appendix B.1.

Note that we obtain a bit  $b$  directly in the clear. This means that when zero testing multiple coefficients at once, we cannot keep an  $n$ -shared bit  $b$  and high-order combine the results of individual zero-testing. Therefore, to test multiple coefficients at once, we will have to proceed differently (see Section 4).

**Complexity.** For simplicity we ignore the reductions modulo  $q$  in the operation count. The complexity of `LinearRefreshMask` is  $3(n - 1)$  operations. We obtain:

$$T_{\text{ZeroTestMult}}(n) = n \cdot (1 + n + 3(n - 1)) + n = n \cdot (4n - 1) \simeq 4n^2$$

The technique has therefore complexity  $\mathcal{O}(n^2)$  for a single coefficient. That is, as opposed to the `ZeroTestAB` algorithm, the complexity is independent from the size of the modulus  $q$ , assuming that arithmetic operations in  $\mathbb{Z}_q$  take unit time. We will see in Section 3.3 that for zero testing a single coefficient, the technique is much faster.



---

**Algorithm 1** ZeroTestMult

---

**Input:**  $x_1, \dots, x_n \in \mathbb{Z}_q$  for prime  $q$ .

**Output:**  $b \in \{0, 1\}$  with  $b = 1$  if  $\sum_i x_i = 0 \pmod{q}$  and  $b = 0$  otherwise

```
1:  $(B_1, \dots, B_n) \leftarrow (x_1, \dots, x_n)$ 
2: for  $j = 1$  to  $n$  do
3:    $u_j \leftarrow \mathbb{Z}_q^*$ 
4:    $(B_1, \dots, B_n) \leftarrow (u_j \cdot B_1 \pmod{q}, \dots, u_j \cdot B_n \pmod{q})$ 
5:    $(B_1, \dots, B_n) \leftarrow \text{LinearRefreshMasks}(B_1, \dots, B_n)$ 
6: end for
7:  $B \leftarrow B_1 + \dots + B_n \pmod{q}$ 
8: if  $B = 0$  then return 1
9: else return 0
```

---

**Security.** The following theorem shows that the adversary does not get more information than whether  $x = 0$  or not. The argument is as follows: if the adversary has at most  $n - 1$  probes, then at least one multiplication by  $u_i \in \mathbb{Z}_q^*$  and subsequent mask refreshing has not been probed. In that case, all output shares of the corresponding mask refreshing can be perfectly simulated, knowing the output bit  $b$ . Namely if  $x \neq 0$ , the output shares must encode a random element in  $\mathbb{Z}_q^*$  (thanks to the multiplication by the random  $u_i \in \mathbb{Z}_q^*$  which has not been probed), and if  $x = 0$ , the output shares are an encoding of 0. In both cases, since by assumption the mask refreshing has not been probed, we can provide a perfect simulation of all output shares of the mask refreshing, which is easily propagated to the end of the algorithm, and eventually the recombination of the shares and the bit  $b$ . We provide the proof in Appendix C.9.

**Theorem 1 (( $n-1$ )-Nlo of ZeroTestMult).** *The ZeroTestMult takes as input  $n$  arithmetic shares  $x_i$  for  $1 \leq i \leq n$  and outputs a bit  $b$  with  $b = 1$  if  $\sum_{i=1}^n x_i = 0 \pmod{q}$  and  $b = 0$  otherwise. Any  $t$  probes can be perfectly simulated from  $x_{|I}$  and  $b$ , with  $|I| \leq t$ .*

### 3.2 Zero testing modulo a prime $q$ via exponentiation

Our second technique also works for prime  $q$  only. It consists in computing

$$b = 1 - x^{q-1} \pmod{q} \quad (1)$$

By Fermat's little theorem, we obtain  $b = 1$  if  $x = 0 \pmod{q}$  and  $b = 0$  otherwise, as required. Given as input the shares  $x_i$  of  $x = x_1 + \dots + x_n \pmod{q}$ , the exponentiation  $x^{q-1} \pmod{q}$  in (1) can be computed with a square-and-multiply, using a sequence of high-order multiplications modulo  $q$ . Eventually we obtain an arithmetic sharing of  $b = b_1 + \dots + b_n \pmod{q}$ , and we recombine the shares to get the bit  $b$ . The complexity of each high-order multiplication modulo  $q$  is  $\mathcal{O}(n^2)$  for  $n$  shares. Hence the complexity is  $\mathcal{O}(n^2 \cdot \log q)$ , assuming that arithmetic operations modulo  $q$  take unit time.

We recall in Appendix C.7 the secure multiplication algorithm SecMult, already considered in [SPOG19]. We then provide in Appendix C.8 the pseudo-code of the ZeroTestExpo algorithm computing the bit  $b$  as in (1). We provide the proof of the following theorem in Appendix C.10.

**Theorem 2 (( $n-1$ )-Nlo of ZeroTestExpo).** *The ZeroTestExpo algorithm is ( $n-1$ )-Nlo.*

### 3.3 Comparison of zero-test algorithms

We provide below a comparison of the 3 zero-test algorithms that work modulo  $q$ , with  $q = 3329$  as in Kyber. We see in Table 3 that for testing a single value, `ZeroTestMult` is more than one order of magnitude faster than `ZeroTestAB` and `ZeroTestExpo`.

	Security order $t$								
	1	2	3	4	5	6	8	10	12
<code>ZeroTestAB</code>	439	1 393	2 593	4 514	6 681	9 289	15 913	24 386	34 428
<code>ZeroTestExpo</code>	182	474	900	1 460	2 154	2 982	5 040	7 634	10 764
<code>ZeroTestMult</code>	14	33	60	95	138	189	315	473	663

Table 3: Operation count for zero testing with arithmetic masking modulo  $q$ , with  $n = t + 1$  shares and  $q = 3329$ .

## 4 High-order polynomial comparison

In this section we extend the zero-testing techniques from Section 3 to multiple coefficients all at once. We refer to Table 1 in Section 1 for a summary of the resulting algorithms and their complexity.

To zero-test a set of Boolean masked coefficients (`PolyZeroTestBool`), we simply perform a sequence of high-order `And`s of the complement in  $\{0, 1\}^k$ , followed by a final zero-testing to get a single bit. The approach is the same for zero testing multiple coefficients arithmetically masked modulo  $2^k$  or a prime  $q$ , thanks to arithmetic to Boolean conversion (`PolyZeroTestAB`). We describe the two algorithms in appendices D.1 and D.2.

When working modulo a prime  $q$ , it is very advantageous to first apply the technique from [BDH<sup>+</sup>21] that reduces the zero-testing of  $\ell$  coefficients to the zero-testing of  $\kappa \ll \ell$  coefficients. If all coefficients are 0, each linear combination will be 0. If at least one of the coefficients is non-zero, the linear combination will be non-zero, except with probability  $1/q$ . Therefore, by using  $\kappa$  linear combinations, we can decrease the error probability to  $q^{-\kappa}$ . To get error probability lower than  $2^{-\lambda}$ , we can therefore take  $\kappa = \lceil \lambda / \log_2 q \rceil$  for security parameter  $\lambda$ . We recall the technique in Appendix D.3. The main advantage is that coefficients of the linear combinations can be computed in the clear, which implies that the complexity of this first step is only  $\mathcal{O}(n)$ .

The remaining  $\kappa$  coefficients must then be zero tested all at once. For this one can use the zero-testing based on multiplicative masking (`ZeroTestMult`), or the zero-testing based on exponentiation (`ZeroTestExpo`). When using the `ZeroTestExpo` algorithm, we keep the resulting bit of each individual zero-test in shared form, so that we can combine them by high-order multiplication using the `SecMult` algorithm. Eventually we recombine the shares to get the result of the global zero-test.

However, when the zero-testing is based on multiplicative masking (`ZeroTestMult`), we obtain the bit  $b$  of an individual zero-testing in the clear, so we must proceed differently. Before applying the zero-testing, we first compute random linear combinations as in [BDH<sup>+</sup>21], but this time the coefficients of the linear combination must be masked with  $n$  shares. For each linear combination, we perform a zero-test of the result. As previously, by repeating the procedure  $\kappa$  times, we can decrease the error probability to  $2^{-\lambda}$  with  $\kappa = \lceil \lambda / \log_2 q \rceil$ .

These last two methods (`PolyZeroTestExpo` and `PolyZeroTestMult`) both work modulo a prime  $q$  only, so it is interesting to compare their complexities (see Table 1). We see that the exponentiation method is faster for small  $q$ , when  $\log_2 q \ll \sqrt{\lambda}$ . Otherwise, the multiplicative masking method is faster. For the Kyber scheme, with  $q = 3329$  and targeting  $\lambda = 128$  bits of security, we expect the two methods to have a similar level of efficiency, and in practice their running time is surprisingly close.

#### 4.1 Polynomial comparison modulo $q$ via multiplicative masking

As explained previously, when zero testing a value  $x$  modulo  $q$  using the multiplicative masking technique (Section 3.1), we obtain the resulting bit  $b$  in the clear, so we cannot zero-test the coefficients iteratively. Instead, we first compute a random linear combination of the individual coefficients modulo  $q$ , and we then perform a zero-test of the result. This approach is similar to [BDH<sup>+</sup>21], except that we must compute the coefficients  $a^{(j)}$  in the linear combination in  $n$ -shared form, as otherwise this can leak information on the input coefficients and then cause a CCA attack.

As previously, we consider as input an arithmetic masking of  $\ell$  coefficients  $x^{(j)}$ , that is  $x^{(j)} = x_1^{(j)} + \dots + x_n^{(j)} \pmod{q}$  for all  $1 \leq j \leq \ell$ . We first apply the reduction algorithm from  $\ell$  to  $\kappa$  coefficients (see Appendix D.3). In the second step, we must therefore zero-test the set of coefficients  $y^{(j)}$  with arithmetic shares  $y_i^{(j)}$  modulo  $q$ , that is  $y^{(j)} = y_1^{(j)} + \dots + y_n^{(j)} \pmod{q}$  for all  $1 \leq j \leq \kappa$ .

For this, we generate random coefficients  $a^{(j)} \in \mathbb{Z}_q$ , and we high-order compute the linear combination:

$$z = \sum_{j=1}^{\kappa} a^{(j)} \cdot y^{(j)} \pmod{q} \quad (2)$$

If  $y^{(j)} = 0$  for all  $1 \leq j \leq \kappa$ , then  $z = 0$ . If  $y^{(j)} \neq 0$  for some  $1 \leq j \leq \kappa$ , then we have  $z \neq 0$ , except with probability  $1/q$ . We can therefore perform a zero-test of  $z$ . The procedure can be repeated a small number of times to have a negligible probability of error. Namely, for  $\kappa$  repetitions with randomly generated  $a^{(j)}$ , the error probability becomes  $q^{-\kappa}$ .

Equation (2) is high-order computed using the arithmetic shares  $y_i^{(j)}$  of the coefficients  $y^{(j)}$ . Similarly the random coefficients  $a^{(j)}$  are generated via  $n$  random shares  $a_i^{(j)}$  in  $\mathbb{Z}_q$ . This is the main difference with the linear combination used in Appendix D.3 for the reduction step, in which the coefficients are computed in the clear. We stress that this time, the coefficients  $a^{(j)}$  must be computed in  $n$ -shared form, and the multiplication  $a^{(j)} \cdot y^{(j)}$  computed with `SecMult`. Namely, the zero-testing is applied on each linear sum, so without masking the  $a^{(j)}$ 's, an equation over the  $y^{(j)}$ 's could be leaked with fewer than  $n$  probes.

From the high-order computation of (2), we obtain the  $n$  shares  $z_i$  of a linear combination  $z$ . We then apply the zero-test procedure from Section 3.1 on the shares  $z_i$ , which outputs a bit  $b$  such that  $b = 1$  if  $z = 0$  and  $b = 0$  otherwise. The procedure is repeated  $\kappa$  times, and if we always obtain  $b = 1$  from the zero-test, we output 1, otherwise we output 0. We provide in Appendix D.4 a pseudocode description of the corresponding algorithm `PolyZeroTestMult`.

For security level  $\lambda$ , the error probability must satisfy  $q^{-\kappa} \leq 2^{-\lambda}$ , so we can take  $\kappa = \lceil \lambda / \log_2 q \rceil$  repetitions. In the second step, taking as input  $\kappa$  coefficients, each linear sum computation and final zero-test has complexity  $\mathcal{O}(\kappa n^2)$ . Therefore, the complexity of the second step is  $\mathcal{O}(\kappa^2 n^2)$ . The total complexity is therefore  $\mathcal{O}(\kappa \ell n + \kappa^2 n^2)$ . We refer to Appendix D.4 for a

more precise operation count. Theorems 3 and 4 below prove the soundness and security of the algorithm respectively; we refer to Appendix D.5 and D.6 for the proofs.

**Theorem 3 (Soundness).** *The PolyZeroTestMult outputs the correct answer, except with probability at most  $q^{-\kappa}$ .*

**Theorem 4 (( $n-1$ )-Nlo of PolyZeroTestMult).** *The PolyZeroTestMult algorithm is ( $n-1$ )-Nlo.*

## 4.2 Polynomial comparison modulo $q$ via exponentiation

In this section, we extend the technique from Section 3.2 to zero test multiple coefficients at once with the exponentiation method. As previously, we are given as input  $\ell$  coefficients  $x^{(j)} \in \mathbb{Z}_q$  with arithmetic shares  $x_i^{(j)}$  modulo a prime  $q$ , and we must output a bit  $b = 1$  if  $x^{(j)} = 0$  for all  $1 \leq j \leq \ell$ , and  $b = 0$  otherwise. We first apply the reduction algorithm, so there remains only  $\kappa$  coefficients  $y^{(j)}$  to be zero-tested, from their arithmetic shares  $y_i^{(j)}$  modulo  $q$ , that is  $y^{(j)} = y_1^{(j)} + \dots + y_n^{(j)} \pmod{q}$  for all  $1 \leq j \leq \kappa$ .

To perform a zero test of all coefficients  $y^{(j)}$  at once, we high-order compute:

$$b = \prod_{j=1}^{\ell} \left(1 - (y^{(j)})^{q-1}\right) \pmod{q} \quad (3)$$

and we obtain  $b = 1$  if  $y^{(j)} = 0$  for all  $1 \leq j \leq \ell$ , and  $b = 0$  otherwise, as required. As previously, Equation (3) can be securely computed by a sequence of high-order multiplication SecMult, using as input the arithmetic shares  $y_i^{(j)}$  modulo  $q$ . The shares of  $b$  are only recombined at the end, so that an adversary with at most  $t = n - 1$  probes does not learn more than the bit  $b$ . Since the complexity of a single SecMult is  $\mathcal{O}(n^2)$ , the complexity for  $\kappa$  coefficients is  $\mathcal{O}(\kappa n^2 \log q)$ , and the total complexity is therefore  $\mathcal{O}(\ell \kappa n + \kappa n^2 \log q)$ .

We provide in Appendix D.7 a pseudocode description of the corresponding PolyZeroTestExpo algorithm. The proof of the following theorem is straightforward and is therefore omitted.

**Theorem 5 (( $n-1$ )-Nlo of PolyZeroTestExpo).** *The PolyZeroTestExpo algorithm is ( $n-1$ )-Nlo.*

## 4.3 Comparison of polynomial zero-testing

We compare in Table 4 below the operation count between three polynomial comparison techniques. For PolyZeroTestAB we work modulo  $2^k$  with  $k = 13$ , while for PolyZeroTestExpo and PolyZeroTestMult we work modulo  $q = 3329$ . We see that both PolyZeroTestExpo and PolyZeroTestMult are much faster than PolyZeroTestAB. This is because for a large number of coefficients  $\ell$ , the asymptotic complexity of PolyZeroTestExpo and PolyZeroTestMult is  $\mathcal{O}(\ell \cdot n)$ , instead of  $\mathcal{O}(\ell \cdot n^2)$  for PolyZeroTestAB. Namely, the reduction to zero testing  $\kappa \ll \ell$  coefficients with complexity  $\mathcal{O}(n)$  only works for a prime modulus  $q$ . We have also performed a C implementation that confirms these results, see Table 5 below.

zero-testing mod $q$	Security order $t$								
	1	2	3	4	5	6	7	8	9
PolyZeroTestAB	61	378	692	1329	2045	2826	3685	4934	6262
PolyZeroTestExpo	44	64	86	109	134	160	188	217	248
PolyZeroTestMult	43	63	83	104	126	149	172	197	223

Table 4: Operation count for polynomial zero testing with arithmetic masking modulo  $q$ , with  $n = t + 1$  shares,  $\ell = 768$  coefficients, in thousands of operation, with  $q = 2^{13}$  for PolyZeroTestAB and  $q = 3329$  for PolyZeroTestExpo and PolyZeroTestMult. We use  $\kappa = 11$ , in order to reach 128 bits of security.

zero-testing mod $q$	Security order $t$								
	1	2	3	4	5	6	7	8	9
PolyZeroTestAB	221	358	503	991	1517	1972	2425	3211	4125
PolyZeroTestMult	83	97	121	164	194	242	268	317	403
PolyZeroTestExpo	79	96	116	153	185	224	254	280	345

Table 5: Running time in thousands of cycles for a C implementation on Intel(R) Core(TM) i7-1065G7, for the same parameters as in Table 4.

zero-testing mod $q$	Security order $t$								
	1	2	3	4	5	6	7	8	9
PolyZeroTestAB	12 297	34 587	69 174	143 706	230 535	327 357	436 476	593 988	763 797
PolyZeroTestMult	8 734	9 020	9 310	9 881	10 739	11 599	12 461	13 601	15 033
PolyZeroTestExpo	8 856	9 119	9 429	9 826	10 408	10 936	11 555	12 215	13 601

Table 6: Number of calls to the rand() function (outputting a 32-bit value), for the same parameters as in Table 4.

## 5 Polynomial comparison for Kyber

In this section, we focus on the polynomial comparison in Kyber [BDK<sup>+</sup>18]. We will recall in Section 6 the full Kyber algorithm, and then describe a complete high-order masking of Kyber.

Recall that computations in Kyber are performed in  $R_q = \mathbb{Z}_q[X]/(X^N + 1)$  with  $N = 256$  and  $q = 3329$ . To reduce the ciphertext size, the coefficients of the ciphertext are compressed from modulo  $q$  to  $d$  bits using the function:

$$\text{Compress}_{q,d}(x) := \left\lfloor (2^d/q) \cdot x \right\rfloor \bmod 2^d$$

and are decompressed using the function  $\text{Decompress}_{q,d}(c) := \lfloor (q/2^d) \cdot c \rfloor$ , with  $d = d_u = 10$  for the first part of the ciphertext, and  $d = d_v = 4$  for the second part, according to the Kyber768 parameters (see Table 7 below).

	$N$	$k$	$q$	$\eta_1$	$\eta_2$	$(d_u, d_v)$	$\delta$
Kyber512	256	2	3329	3	2	(10,4)	$2^{-139}$
Kyber768	256	3	3329	2	2	(10,4)	$2^{-164}$
Kyber1024	256	4	3329	2	2	(11,5)	$2^{-174}$

Table 7: Parameter sets for Kyber.

In the IND-CCA decryption algorithm based on the Fujisaki-Okamoto transform [FO99], we must perform a polynomial comparison between two compressed ciphertexts: the input ciphertext  $\tilde{c}$ , and the re-encrypted ciphertext  $c$ . The  $\text{Compress}_{q,d}$  function is applied coefficient-wise, so for simplicity we first consider a single coefficient. Let  $x$  be the re-encrypted coefficient modulo  $q$  before compression, and let  $c$  be the resulting compressed coefficient, that is  $c = \text{Compress}_{q,d}(x)$ . We must therefore perform the comparison with the input ciphertext  $\tilde{c}$  modulo  $2^d$ :

$$\tilde{c} \stackrel{?}{=} \text{Compress}_{q,d}(x) \pmod{2^d} \quad (4)$$

There are two possible approaches to perform this comparison. The first approach consists in performing the comparison as in (4). Since the re-encrypted coefficient  $x$  is arithmetically masked modulo  $q$ , we show how to high-order compute  $\text{Compress}_{q,d}(x)$  with arithmetically masked input modulo  $q$ , and Boolean masked output in  $\{0, 1\}^d$ . We can then perform the high-order polynomial comparison over Boolean shares, using the `PolyZeroTestBool` algorithm (see Appendix D.1). We describe in Section 5.1 such high-order computation of the `Compress` function.

A second approach is to avoid the computation of the `Compress` function, as in [BGR<sup>+</sup>21]. Namely instead of performing the comparison over  $\{0, 1\}^d$  as in (4), one can equivalently compute the set of possible candidates  $\tilde{x}_i$  such that  $\tilde{c} = \text{Compress}_{q,d}(\tilde{x}_i)$ . One must then determine whether the re-encrypted coefficient  $x$  is equal to one of the (public) candidates  $\tilde{x}_i$ , using the  $n$  arithmetic shares of  $x$  modulo  $q$ . Our contribution compared to [BGR<sup>+</sup>21] is to describe an alternative, faster technique when the number of candidates  $\tilde{x}_i$  is small, which is the case for  $d = d_u = 10$  (see Section 5.2).

Finally, we argue that the best approach is hybrid: for the first  $\ell_1 = 768$  coefficients of the ciphertext with  $d_u = 10$ , we do not compute the `Compress` function and apply our faster technique for the small number of candidates  $\tilde{x}_i$ , and for the remaining  $\ell_2 = 256$  coefficients with  $d_v = 4$ , we high-order compute the `Compress` function. We describe this hybrid approach in Section 5.3.

## 5.1 High-order computation of the `Compress` function

We provide the first description of the high-order computation of the `Compress` function of Kyber. Our technique can be seen as a generalization of the first-order technique of [FBR<sup>+</sup>21], based on modulus switching: it consists in first using more precision, so that the error induced by the modulus switching can be completely eliminated, after a logical shift.

The `Compress` function is defined as:

$$\text{Compress}_{q,d}(x) = \left\lfloor \frac{2^d \cdot x}{q} \right\rfloor \pmod{2^d}$$

We are given as input an arithmetic sharing of  $x = x_1 + \dots + x_n \pmod{q}$  and we want to compute a Boolean sharing of  $y = \text{Compress}_{q,d}(x) = y_1 \oplus \dots \oplus y_n \in \{0, 1\}^d$ . Note that in [BGR<sup>+</sup>21], the

authors only described the high-order masking of the `Compress` function with 1-bit output, which corresponds to the IND-CPA decryption function of Kyber. Here we high-order mask `Compress` for any number of output bits  $d$  (for example  $d = d_u = 10$  or  $d = d_v = 4$  in Kyber768). For the special case  $d = 1$  there are more efficient techniques, see for example [BGR<sup>+</sup>21] and [CGMZ22].

We proceed as follows. We first perform a modulus switching of the input coefficients  $x_i$  but with more precision; that is we work modulo  $2^{d+\alpha}$  for some parameter  $\alpha > 0$  and compute:

$$z_1 = \left\lfloor \frac{x_1 \cdot 2^{d+\alpha}}{q} \right\rfloor + 2^{\alpha-1} \bmod 2^{d+\alpha}, \quad z_i = \left\lfloor \frac{x_i \cdot 2^{d+\alpha}}{q} \right\rfloor \bmod 2^{d+\alpha} \quad \text{for } 2 \leq i \leq n$$

The rounding can be computed by writing:

$$\left\lfloor \frac{x_i \cdot 2^{d+\alpha}}{q} \right\rfloor = \left\lfloor \frac{x_i \cdot 2^{d+\alpha}}{q} + \frac{1}{2} \right\rfloor = \left\lfloor \frac{x_i \cdot 2^{d+\alpha+1} + q}{2q} \right\rfloor$$

which is the quotient of the Euclidean division of  $x_i \cdot 2^{d+\alpha+1} + q$  by  $2q$ .

We then perform an arithmetic to Boolean conversion of the arithmetic shares  $z_1, \dots, z_n$ , followed by a logical shift by  $\alpha$  bits. This can be done with complexity  $\mathcal{O}((d + \alpha) \cdot n^2)$  using [CGV14]. By definition we obtain:

$$y_1 \oplus \dots \oplus y_n = \left\lfloor \left( \sum_{i=1}^n z_i \right) / 2^\alpha \right\rfloor \pmod{2^d} \quad (5)$$

and eventually we output the Boolean shares  $y_1, \dots, y_n$ . We show below that we indeed have  $\text{Compress}_{q,d}(x) = y_1 \oplus \dots \oplus y_n$  as required, under the condition  $2^\alpha > q \cdot n$ . This condition determines the number  $\alpha$  of bits of precision as a function of the number of shares  $n$ . We provide the pseudocode in Algorithm 2 below.

---

**Algorithm 2** HOCompress

---

**Input:**  $x_1, \dots, x_n \in \mathbb{Z}_q$

**Output:**  $y_1, \dots, y_n \in \{0, 1\}^d$  such that  $y_1 \oplus \dots \oplus y_n = \text{Compress}_{q,d}(x_1 + \dots + x_n)$

- 1:  $\alpha \leftarrow \lceil \log_2(q \cdot n) \rceil$
  - 2:  $z_1 \leftarrow \lfloor (x_1 \cdot 2^{d+\alpha+1} + q) / (2q) \rfloor + 2^{\alpha-1} \bmod 2^{d+\alpha}$
  - 3: **for**  $i = 2$  to  $n$  **do**  $z_i \leftarrow \lfloor (x_i \cdot 2^{d+\alpha+1} + q) / (2q) \rfloor \bmod 2^{d+\alpha}$
  - 4:  $(c_1, \dots, c_n) \leftarrow \text{ArithmeticToBoolean}(d + \alpha, (z_1, \dots, z_n))$
  - 5: **for**  $i = 1$  to  $n$  **do**  $y_i \leftarrow c_i \gg \alpha$
  - 6: **return**  $y_1, \dots, y_n$
- 

**Theorem 6 (Soundness).** *Given  $x_1, \dots, x_n \in \mathbb{Z}_q$  as input for odd  $q \in \mathbb{N}$ , the algorithm HOCompress computes  $y_1, \dots, y_n \in \{0, 1\}^d$  such that  $y_1 \oplus \dots \oplus y_n = f(x_1 + \dots + x_n)$  where  $f(x) = \lfloor x \cdot 2^d / q \rfloor \bmod 2^d$ .*

*Proof.* Given  $x \in \mathbb{Z}_q$ , we have:

$$f(x) = \left\lfloor \frac{x \cdot 2^d}{q} \right\rfloor \bmod 2^d = \left\lfloor \frac{x \cdot 2^d}{q} + \frac{1}{2} \right\rfloor \bmod 2^d = \left\lfloor \frac{x \cdot 2^{d+1} + q}{2q} \right\rfloor \bmod 2^d$$

We write the Euclidean division  $x \cdot 2^{d+1} + q = y \cdot (2q) + \delta$  with  $y, \delta \in \mathbb{Z}$  and  $0 \leq \delta < 2q$ . Therefore  $f(x) = y \pmod{2^d}$ . Moreover we must have  $\delta \neq 0$ , since otherwise  $x = 0 \pmod{q}$ , which gives  $y = 1/2$ , a contradiction. Therefore  $0 < \delta < 2q$ .

In the following we compute the modular reduction over  $\mathbb{Q}$ . We have for some  $|e| \leq n/2$ :

$$\begin{aligned} \sum_{i=1}^n z_i &= \sum_{i=1}^n \left\lfloor \frac{x_i \cdot 2^{d+\alpha}}{q} \right\rfloor + 2^{\alpha-1} = \sum_{i=1}^n \frac{x_i \cdot 2^{d+\alpha}}{q} + 2^{\alpha-1} + e \pmod{2^{d+\alpha}} \\ &= \frac{x \cdot 2^{d+\alpha}}{q} + 2^{\alpha-1} + e = 2^\alpha \cdot \frac{x \cdot 2^{d+1} + q}{2q} + e \pmod{2^{d+\alpha}} \\ &= 2^\alpha \cdot \left( y + \frac{\delta}{2q} \right) + e = 2^\alpha \cdot y + 2^\alpha \left( \frac{\delta}{2q} + e \cdot 2^{-\alpha} \right) \pmod{2^{d+\alpha}} \end{aligned}$$

From (5), if we ensure that  $0 \leq \delta/(2q) + e \cdot 2^{-\alpha} < 1$ , then we must have  $y = f(x) = y_1 \oplus \dots \oplus y_n$  as required. Since  $0 < \delta < 2q$ , it is sufficient to ensure that  $e \cdot 2^{-\alpha} < 1/(2q)$ , and therefore a sufficient condition is  $n \cdot 2^{-\alpha} < 1/q$ . Therefore it is sufficient to ensure  $2^\alpha > q \cdot n$ .  $\square$

**Complexity of HOCompress.** The number of operations of the HOCompress algorithm above is  $T_{\text{HOComp}}(n, d, q) = 5n + 1 + T_{\text{AB}}(d + \alpha, n)$ . We refer to [CGV14] for the operation count of arithmetic to Boolean conversion, with  $T_{\text{AB}}(d + \alpha, n) = \mathcal{O}((d + \alpha) \cdot n^2)$ . With  $d < \log_2 q$  and  $\alpha = \lceil \log_2(q \cdot n) \rceil$ , the total complexity of HOCompress is therefore  $\mathcal{O}(n^2 \cdot (\log q + \log n))$ .

**Security.** The following theorem shows that the HOCompress achieves the  $(n - 1)$ -NI property. The proof follows from the  $(n - 1)$ -NI property of the ArithmeticToBoolean algorithm, and the fact that the perfect simulation of  $z_i$  requires the knowledge of the input  $x_i$  only.

**Theorem 7 (( $n - 1$ )-NI security).** *The HOCompress algorithm achieves the  $(n - 1)$ -NI property.*

**Polynomial comparison with Compress.** Recall that we must perform the comparison  $\tilde{c} \stackrel{?}{=} \text{Compress}_{q,d}(x)$ , where for simplicity we consider a single coefficient  $\tilde{c}$ . By applying the HOCompress algorithm, we obtain  $n$  Boolean shares such that  $c = c_1 \oplus \dots \oplus c_n$ . We must therefore zero-test the value  $(c_1 \oplus \tilde{c}) \oplus c_2 \oplus \dots \oplus c_n$ , which can be done using the ZeroTestBool algorithm from Appendix C.3.

For multiple coefficients, we apply the HOCompress algorithm separately on each coefficient  $x^{(j)}$  of the re-encrypted uncompressed ciphertext. We obtain the compressed ciphertext  $c$  masked with  $n$  Boolean shares. As previously, we xor each coefficient of the input ciphertext  $\tilde{c}$  with the first share of the corresponding coefficient in  $c$ , and we apply the PolyZeroTestBool algorithm from Appendix D.1 to perform the comparison.

## 5.2 Polynomial comparison for Kyber without Compress

In this section we consider an alternative approach for ciphertext comparison, already used in [BGR<sup>+</sup>21], that performs the comparison on uncompressed ciphertexts, in order to avoid the high-order computation of the  $\text{Compress}_{q,d}(x)$  function as above. Under this approach, we describe a more efficient technique for ciphertext comparison for  $d = d_u = 10$ , which is the case for 3/4 of the polynomial coefficients in Kyber.



For simplicity, we first consider a single polynomial coefficient. Given a compressed input ciphertext  $\tilde{c}$  and an uncompressed re-encrypted ciphertext  $x$ , we must check that  $\tilde{c} = \text{Compress}_{q,d}(x)$ , where  $x$  is arithmetically masked with  $n$  shares modulo  $q$ . For this we use the equivalence:

$$\tilde{c} = \text{Compress}_{q,d}(x) \iff x \in \text{Compress}_{q,d}^{-1}(\tilde{c})$$

Given  $\tilde{c}$  as input, we must therefore compute the (public) list of candidates  $\text{Compress}_{q,d}^{-1}(\tilde{c})$ , which corresponds to a certain interval in  $\mathbb{Z}_q$ , and check whether  $x$  belongs to this interval. For this, the authors of [BGR<sup>+</sup>21] describe a high-order algorithm that performs two high-order comparisons with the interval bounds. We recall the corresponding `RangeTestShares` algorithm in Appendix E.3, with the pseudo-code and the operation count.

However, we observe that when the number of candidates is small (which is the case for  $d = d_u = 10$ ), it is more efficient to perform individual comparisons. More precisely, letting  $\{\tilde{x}_1, \dots, \tilde{x}_m\} = \text{Compress}_{q,d}^{-1}(\tilde{c})$  be the list of candidates, we must test that  $x = \tilde{x}_i$  for some  $1 \leq i \leq m$ . Recall that  $x$  is arithmetically masked with  $n$  shares modulo  $q$ . Therefore we can high-order compute  $z = \prod_{i=1}^m (x - \tilde{x}_i) \bmod q$  and then apply a high-order zero-test of  $z$  modulo  $q$ . We describe the technique below.

**Computing the set of candidates.** Given a compressed coefficient  $\tilde{c}$ , we must compute the list of candidates  $\text{Compress}_{q,d}^{-1}(\tilde{c})$ . While such preimage can be easily tabulated for all its  $2^d$  possible inputs (as done in [BGR<sup>+</sup>21]), in the following we describe a concrete algorithm. This can be useful for embedded applications with limited memory.

From [BDK<sup>+</sup>18], we know that for any  $x \in \mathbb{Z}_q$  such that  $\tilde{c} = \text{Compress}_{q,d}(x)$ , letting  $y = \text{Decompress}_{q,d}(\tilde{c}) = \lfloor (q/2^d) \cdot \tilde{c} \rfloor$  we must have:

$$|y - x \bmod^{\pm} q| \leq B_{q,d} := \left\lfloor \frac{q}{2^{d+1}} \right\rfloor$$

Therefore the number of candidates is upper-bounded by  $2B_{q,d} + 1$ ; see Table 8 for the value of the upper-bound, and the maximum number of candidates, for  $q = 3329$ .

	$d = 4$	$d = 5$	$d = 10$	$d = 11$
$2 \cdot B_{q,d} + 1$	209	105	5	3
$N_{\max}$	209	105	4	2

Table 8: Upper-bound on the number of candidates, and maximum number of candidates  $N_{\max}$ , for  $q = 3329$ .

The following lemma shows that there are always at least  $2B_{q,d} - 1$  candidates around the decompressed value  $y$ , with possibly 2 additional candidates. We can then test these 2 candidates by applying the `Compress` function. Given  $0 \leq a < b < q$ , we denote by  $[a, b]_q$  the discrete interval  $\{a, a + 1, \dots, b\}$ ; similarly given  $0 \leq b < a < q$ , we denote by  $[a, b]_q$  the discrete interval  $\{a, a + 1, \dots, q - 1, 0, 1, \dots, b\}$ . In Appendix E.1, we provide the proof of the following Lemma, from which we derive a concrete algorithm.

**Lemma 1.** *Assume  $d < \lceil \log_2 q \rceil$ . Let  $\tilde{c} \in \mathbb{Z}_{2^d}$  and let  $y = \text{Decompress}_{q,d}(\tilde{c})$ . We have  $[y - B_{q,d} + 1, y + B_{q,d} - 1]_q \subset \text{Compress}_{q,d}^{-1}(\tilde{c}) \subset [y - B_{q,d}, y + B_{q,d}]_q$ .*

**Individual comparisons.** We see in Table 8 that the number of candidates is small for  $d = 10, 11$ , so we describe an alternative algorithm to the range test performed in [BGR<sup>+</sup>21], based on individual comparisons. Letting  $\{\tilde{x}_1, \dots, \tilde{x}_m\} = \text{Compress}_{q,d}^{-1}(\tilde{c})$  be the list of candidates, we must test whether  $x = \tilde{x}_i$  for some  $1 \leq i \leq m$ . For this, given an arithmetically masked  $x$  with  $n$  shares with  $x = x_1 + \dots + x_n \pmod{q}$ , we high-order compute the value

$$z = \prod_{i=1}^m (x - \tilde{x}_i) \pmod{q} \quad (6)$$

and we have that  $z = 0 \pmod{q}$  if and only if  $x = \tilde{x}_i$  for some  $1 \leq i \leq m$ . We provide in Appendix E.2 the pseudocode description of the corresponding `SecMultList` algorithm, and its proof of security against  $t = n - 1$  probes. As a second step, one can then apply a high-order zero-test of  $z$  modulo  $q$ , either the `ZeroTestMult` algorithm from Section 3.1, or the `ZeroTestExpo` algorithm from Section 3.2.

The above applies for a single coefficient  $x$ . In reality we must compare  $\ell$  coefficients, so for each coefficient  $x^{(j)}$  whose compressed value must be compared to the coefficient  $\tilde{c}_j$  of the input ciphertext  $\tilde{c}$ , we compute the corresponding list of candidates from  $\tilde{c}_j$ , and then the corresponding arithmetically masked  $z^{(j)}$ , which must all be equal to 0. Therefore a polynomial zero-test is applied to the set of arithmetically masked  $z^{(j)}$ 's modulo  $q$ , either the `PolyZeroTestMult` algorithm from Section 4.1, or the `PolyZeroTestExpo` algorithm from Section 4.2.

### 5.3 Ciphertext comparison in Kyber: hybrid approach

We first compare in Table 9 the efficiency of the approaches with and without `Compress`. Since the number of candidates is small for the coefficients with compression parameter  $d_u = 10$  bits, without using the `Compress` function we can use either the `RangeTestShares` algorithm from [BGR<sup>+</sup>21], or our `SecMultList` algorithm. We see in Table 9 that the latter is significantly faster. It is also faster than applying the `Compress` function with our `HCompress` algorithm. On the other hand, for the coefficients with compression to  $d_v = 4$  bits, without using the `Compress` function, one must use the `RangeTestShares` algorithm from [BGR<sup>+</sup>21]. But we see that our `HCompress` is nevertheless faster. The reason is that it uses only a single arithmetic to Boolean conversion with a power-of-two modulus, whereas `RangeTestShares` uses two arithmetic to Boolean conversions, moreover modulo  $q$ , which is more costly than with power-of-two moduli.

In summary, from Table 9, we deduce that for  $d = d_u = 10$ , our `SecMultList` approach without `Compress` is faster, while for  $d = d_v = 4$ , our `HCompress` algorithm is faster. Therefore, to perform the ciphertext comparison in Kyber, we use a hybrid approach, applying the `Compress` function only for the last  $\ell_2 = 256$  coefficients of the ciphertext, for which  $d = d_v = 4$ .

		Security order $t$						
		1	2	3	4	5	6	7
$d_u = 10$	<code>RangeTestShares</code> [BGR <sup>+</sup> 21]	707	2 318	4 314	7 577	11 225	15 620	20 400
	<code>SecMultList</code>	45	120	230	375	555	770	1 020
	<code>HCompress</code>	131	868	1 579	3 181	4 898	6 764	8 809
$d_v = 4$	<code>RangeTestShares</code> [BGR <sup>+</sup> 21]	707	2 318	4 314	7 577	11 225	15 620	20 400
	<code>HCompress</code>	101	658	1 195	2 431	3 740	5 162	6 721

Table 9: Comparison of the `RangeTestShares`, `SecMultList` and `HCompress` algorithms, in number of operations, for  $q = 3329$  and  $d = d_u = 10$  or  $d = d_v = 4$ .

**Procedure for ciphertext comparison.** We summarize our hybrid approach below. Recall that for masking the IND-CCA decryption of Kyber, we must perform a comparison between the unmasked input ciphertext  $\tilde{c}$ , and the masked re-encrypted ciphertext  $c$ . Moreover, with the Kyber768 parameters, a ciphertext consists of 4 polynomials with 256 coefficients each. The coefficients of the first 3 polynomials are compressed with  $d_u = 10$  bits, while the coefficients of the last polynomial are compressed with  $d_v = 4$  bits. Starting from the re-encrypted uncompressed ciphertext  $c_u$  which is masked modulo  $q$ , and given the input ciphertext  $\tilde{c}$ , we proceed as follows:

1. For each of the first  $\ell_1 = 768$  coefficients of  $c_u$ , with compression parameter  $d_u = 10$ , we use the individual comparison technique from Section 5.2 (Algorithm `SecMultList`). We obtain a set of values  $z^{(j)}$  arithmetically masked modulo  $q$ , that must all be equal to 0, for  $1 \leq j \leq \ell_1$ .
2. For each of the last  $\ell_2 = 256$  coefficients of  $c_u$ , we apply the `HOCCompress` algorithm with  $d_v = 4$  bits. We obtain a set of  $\ell_2$  coefficients  $c^{(j)}$  for  $1 \leq j \leq \ell_2$ , which are Boolean masked with  $n$  shares.
3. We xor each of the last  $\ell_2$  coefficients of the input ciphertext  $\tilde{c}$  to the first Boolean share of each of the corresponding  $\ell_2$  coefficients  $c^{(j)}$ . This gives a vector of  $\ell_2$  coefficients  $x^{(j)}$  for  $1 \leq j \leq \ell_2$ , which are Boolean masked with  $n$  shares, and that must all be equal to 0.
4. We apply the `PolyZeroTestBool` algorithm (Alg. 21) to the set of  $\ell_2$  coefficients  $x^{(j)}$ , but without recombining the shares at the end of the `ZeroTestBoolLog` algorithm. More precisely, we obtain Boolean shares  $b_i$  for  $1 \leq i \leq n$ , with  $b' = b_1 \oplus \dots \oplus b_n$  and  $b' = 1$  if the  $\ell_2$  coefficients  $x^{(j)}$  are zero, and  $b' = 0$  otherwise.
5. We take the complement of  $b'$  by taking the complement of  $b_1$ , and convert the result from Boolean to arithmetic masking modulo  $q$ , using the table-based algorithm from [CGMZ22]. We obtain an additional coefficient  $z^{(\ell_1+1)}$  arithmetically masked modulo  $q$ , and that must be equal to 0.
6. Finally, we perform a zero-test of the  $\ell_1 + 1$  coefficients  $z^{(i)}$  for  $1 \leq i \leq \ell_1 + 1$ , using either the `PolyZeroTestExpo` or the `PolyZeroTestMult` algorithm. We obtain a bit  $b = 1$  if the two ciphertexts are equal, and  $b = 0$  otherwise, as required.

#### 5.4 Operation count and concrete running time

We provide in Table 10 a comparison of the operation count for the ciphertext comparison in Kyber, first using the approach from [BGR<sup>+</sup>21] without `Compress`, and then our hybrid approach described in the previous section, with either the `PolyZeroTestExpo` or the `PolyZeroTestMult` methods. We see that the hybrid approach is significantly faster, especially for high security orders. We have also performed a C implementation that confirms these results, see Table 11 below.

Polynomial comparison in Kyber	Security order $t$							
	1	2	3	4	5	6	7	8
Without <code>Compress</code> [BGR <sup>+</sup> 21]	786	2 574	4 792	8 413	12 465	17 346	22 657	29 770
Hybrid, with <code>PolyZeroTestExpo</code>	121	351	606	1 071	1 584	2 156	2 794	3 650
Hybrid, with <code>PolyZeroTestMult</code>	121	350	603	1 065	1 575	2 144	2 778	3 629

Table 10: Operation count for the three proposed methods to perform ciphertext comparison (with `Compress` and without `Compress` using `PolyZeroTestExpo` or `PolyZeroTestMult`), in thousands of operations.

Polynomial comparison in Kyber	Security order $t$							
	1	2	3	4	5	6	7	8
Without Compress [BGR <sup>+</sup> 21]	1 395	3 722	6 230	9 619	14 517	19 206	24 783	33 675
Hybrid, with PolyZeroTestExpo	191	415	563	914	1 641	2 249	2 745	3 758
Hybrid, with PolyZeroTestMult	185	410	562	966	1 731	2 046	2 829	3 842

Table 11: Running time in thousands of cycles for a C implementation on Intel(R) Core(TM) i7-1065G7, for the three methods considered in Table 10.

## 6 Fully masked implementation of Kyber

Kyber is the lattice-based encryption scheme selected for standardization by the NIST competition [BDK<sup>+</sup>18, ABD<sup>+</sup>21]. Its security is based on the hardness of the module learning-with-errors (M-LWE) problem. The IND-CCA secure key establishment mechanism (KEM) is obtained by applying the Fujisaki-Okamoto transform [FO99, HHK17]. The Kyber submission provides three parameters sets Kyber512, Kyber768 and Kyber1024, with claimed security levels equivalent to AES-128, AES-192 and AES-256 respectively. The three parameter sets share the common parameters  $N = 256$ ,  $q = 3329$  and  $\eta_2 = 2$ , while the security level is defined by setting the module rank  $k = 2, 3, 4$ , and the parameters  $\eta_1, d_t, d_u$  and  $d_v$  (see Table 7).

We refer to Appendix F for an overview of ring-LWE encryption [LPR10]. In the following, we first recall the definition of the Kyber scheme. We then describe the evaluation of the Kyber decapsulation mechanism, secure at any order, using the techniques from the previous sections.

### 6.1 The Kyber Key Encapsulation Mechanism (KEM)

**The Kyber IND-CPA encryption.** The Kyber scheme is based on the module learning-with-errors problem (M-LWE) in module lattices [LS15]. For a modulo rank  $k$ , we use a public random  $k \times k$  matrix  $\mathbf{A}$  with elements in  $\mathcal{R}_q$ . We set  $\chi_\eta$  as the centered binomial distribution with support  $\{-\eta, \dots, \eta\}$ , and extended to the distribution of polynomials of degree  $N$  with entries independently sampled from  $\chi_\eta$ . The public-key is  $\mathbf{t} = \mathbf{A} \cdot \mathbf{s} + \mathbf{e} \in \mathcal{R}_q^k$  and the secret key is  $\mathbf{s}$ , where  $\mathbf{s}, \mathbf{e} \leftarrow \chi_{\eta_1}^k$  for some parameter  $\eta_1$ . To CPA-encrypt a message  $m \in \mathcal{R}$  with binary coefficients, one computes  $(\mathbf{c}_1, c_2) \in \mathcal{R}_q^k \times \mathcal{R}_q$  such that  $\mathbf{c}_1 = \mathbf{A}^T \cdot \mathbf{r} + \mathbf{e}_1$  and  $c_2 = \mathbf{t}^T \cdot \mathbf{r} + e_2 + \lfloor q/2 \rfloor \cdot m$ , where  $\mathbf{r} \leftarrow \chi_{\eta_1}^k$ ,  $\mathbf{e}_1 \leftarrow \chi_{\eta_2}^k$  and  $e_2 \leftarrow \chi_{\eta_2}$ , for some parameter  $\eta_2$ . To decrypt a ciphertext  $(\mathbf{c}_1, c_2)$ , one computes:

$$u = c_2 - \mathbf{s}^T \cdot \mathbf{c}_1 = \mathbf{e}^T \cdot \mathbf{r} + e_2 - \mathbf{s}^T \cdot \mathbf{e}_1 + \lfloor q/2 \rfloor \cdot m \approx \lfloor q/2 \rfloor \cdot m$$

Kyber instantiates the M-LWE-based encryption scheme above with  $N = 256$  and a prime  $q = 3329$ ; see Table 7 for the other parameters. We recall the pseudo-code from [BDK<sup>+</sup>18] below. For simplicity we omit the NTT transform for fast polynomial multiplication. The NTT is indeed a linear operation, so it is easily masked with arithmetic masking modulo  $q$ .

---

#### Algorithm 3 Kyber.CPA.KeyGen()

---

- 1:  $\rho, \sigma \leftarrow \{0, 1\}^{256}$
  - 2:  $\mathbf{A} \leftarrow \mathcal{R}_q^{k \times k} := \text{Sam}(\rho)$
  - 3:  $\mathbf{s}, \mathbf{e} \leftarrow \chi_{\eta_1}^k \times \chi_{\eta_1}^k := \text{Sam}(\sigma)$
  - 4:  $\mathbf{t} := \text{Compress}_{q, d_t}(\mathbf{A}\mathbf{s} + \mathbf{e})$
  - 5: **return**  $pk := (\mathbf{t}, \rho), sk := \mathbf{s}$
-

---

**Algorithm 4** Kyber.CPA.Dec( $sk, c = (\mathbf{u}, v)$ )

---

```
1:  $\mathbf{u} := \text{Decompress}_{q,d_u}(\mathbf{u})$ 
2:  $v := \text{Decompress}_{q,d_v}(v)$ 
3: return  $\text{Compress}_{q,1}(v - \mathbf{s}^T \mathbf{u})$ 
```

---

---

**Algorithm 5** Kyber.CPA.Enc( $pk, m$ )

---

```
1:  $r \leftarrow \{0, 1\}^{256}$ 
2:  $\mathbf{t} := \text{Decompress}_{q,d_t}(\mathbf{t})$ 
3:  $\mathbf{A} \leftarrow \mathcal{R}_q^{k \times k} := \text{Sam}(\rho)$ 
4:  $\mathbf{r}, \mathbf{e}_1, e_2 \leftarrow \chi_{\eta_1}^k \times \chi_{\eta_2}^k \times \chi_{\eta_2} := \text{Sam}(r)$ 
5:  $\mathbf{u} := \text{Compress}_{q,d_u}(\mathbf{A}^T \mathbf{r} + \mathbf{e}_1)$ 
6:  $v := \text{Compress}_{q,d_v}(\mathbf{t}^T \mathbf{r} + e_2 + \lfloor q/2 \rfloor \cdot m)$ 
7: return  $c := (\mathbf{u}, v)$ 
```

---

**The Kyber CCA-secure KEM.** The Kyber scheme provides a CCA-secure key encapsulation mechanism, based on the Fujisaki-Okamoto transform [FO99]. We recall the pseudo-code from [BDK<sup>+</sup>18] below. It requires two different hash functions  $H$  and  $G$ . The main principle of the Fujisaki-Okamoto transform is to check the validity of a ciphertext by performing a re-encryption with the same randomness (see the variable  $r'$  at Line 3 of Algorithm 7 below), and a comparison with the original ciphertext.

Note that the Kyber.Decaps algorithm does not output  $\perp$  for invalid ciphertexts, as originally in the FO transform. Instead, it outputs a pseudo-random value from the hash of a secret seed  $z$  and the ciphertext  $c$ . This variant of the FO transform was proven secure in [HHK17]. However, the variant remains secure even if the adversary is given the result of the ciphertext comparison, under the condition that the IND-CPA scheme is  $\gamma$ -spread, which essentially means that ciphertexts have sufficiently large entropy (see [HHK17]), which is the case in Kyber. Therefore, in the high-order masking of Kyber, the bit  $b$  of the comparison can be computed in the clear (as in [BGR<sup>+</sup>21]), because for the simulation of the probes the bit  $b$  can be given for free to the simulator.

---

**Algorithm 6** Kyber.Encaps( $pk$ )

---

```
1:  $m \leftarrow \{0, 1\}^{256}$ 
2:  $(\hat{K}, r) := G(H(pk), m)$ 
3:  $c := \text{Kyber.CPA.Enc}(pk, m; r)$ 
4:  $K = H(\hat{K}, H(c))$ 
5: return  $c, K$ 
```

---

---

**Algorithm 7** Kyber.Decaps( $sk = (\mathbf{s}, z, \mathbf{t}, \rho), c = (\mathbf{u}, v)$ )

---

```
1:  $m' := \text{Kyber.CPA.Dec}(\mathbf{s}, c)$ 
2:  $(\hat{K}', r') := G(H(pk), m')$ 
3:  $(\mathbf{u}', v') := \text{Kyber.CPA.Enc}((\mathbf{t}, \rho), m'; r')$ 
4: if  $(\mathbf{u}', v') = (\mathbf{u}, v)$  then
   return  $K := H(\hat{K}', H(c))$ 
5: else return  $K := H(z, H(c))$ 
```

---

## 6.2 High-order masking of Kyber

We describe the high-order masking of the Kyber.Decaps algorithm recalled above (Algorithm 7), using the techniques from the previous sections.

1. We consider Line 1 of Algorithm 7, with the IND-CPA decryption as the first step. We assume that the secret key  $\mathbf{s} \in \mathcal{R}^k$  is initially masked with  $n$  shares, with  $\mathbf{s} = \mathbf{s}_1 + \dots + \mathbf{s}_n \pmod{q}$ , where  $\mathbf{s}_i \in (\mathcal{R}_q)^k$  for all  $1 \leq i \leq n$ . Therefore, at Line 3 of the Kyber.CPA.Dec algorithm, we obtain a value  $v - \mathbf{s}^T \mathbf{u}$  that is arithmetically  $n$ -shared modulo  $q$ . We must therefore compute the  $\text{Compress}_{q,1}$  function on this value. For this we use the modulus switching and table recomputation technique from [CGMZ22], which outputs a Boolean masked message  $m' = m_1 \oplus \dots \oplus m_n = \text{th}(v - \mathbf{s}^T \mathbf{u})$ .

2. At Line 2 of Algorithm 7, starting from the Boolean masked  $m'$ , we use an  $n$ -shared Boolean implementation of the hash function  $G$ , and obtain as output the Boolean  $n$ -shared values  $\hat{K}'$  and  $r'$ .
3. At Line 3 of Algorithm 7, we start with Line 4 of Algorithm 5 which is the masked binomial sampling. Starting from the Boolean  $n$ -shared  $r'$ , we must obtain values  $\mathbf{r}$ ,  $\mathbf{e}_1$  and  $e_2$  which are arithmetically  $n$ -shared modulo  $q$ . For this we use the  $n$ -shared binomial sampling from [CGMZ22], based on Boolean to arithmetic modulo  $q$  conversion (based on table recomputation). We use the random generation modulo  $q$  described in Appendix A.
4. We proceed with lines 5 and 6 of Algorithm 5. We obtain the values  $\mathbf{A}^T \cdot \mathbf{r} + \mathbf{e}_1$  and  $\mathbf{t}^T \cdot \mathbf{r} + e_2 + \lfloor q/2 \rfloor \cdot m$  arithmetically  $n$ -shared modulo  $q$ . In particular, the  $n$ -shared value  $\lfloor q/2 \rfloor \cdot m$  is obtained using the table-based Boolean to arithmetic modulo  $q$  conversion from [CGMZ22].
5. At Line 6 of Algorithm 5, the  $n$ -shared value  $\mathbf{t}^T \cdot \mathbf{r} + e_2 + \lfloor q/2 \rfloor \cdot m$  is high-order compressed into  $v'$  using the HOCCompress algorithm from Section 5.1. The value  $v'$  is therefore Boolean  $n$ -shared in  $\{0, 1\}^{d_v}$ . On the other hand, the vector  $\mathbf{u}'$  at Line 5 is left uncompressed.
6. For the ciphertext comparison at Line 4 of Algorithm 7, we use the hybrid technique from Section 5.3 with the arithmetically masked modulo  $q$  uncompressed vector  $\mathbf{u}'$ , and the Boolean masked compressed value  $v'$ . We obtain a bit  $b$  in the clear.
7. Finally, if  $b = 1$ , we use the Boolean  $n$ -shared  $\hat{K}'$  to obtain a Boolean  $n$ -shared session key  $K$ , using an  $n$ -shared implementation of  $H$ . Similarly, if  $b = 0$ , we use the Boolean  $n$ -shared secret  $z$  to obtain the Boolean  $n$ -shared session key  $K$ .

## 7 Fully masked implementation of Saber

Saber [BMD<sup>+</sup>21] is based on the hardness on the module learning-with-rounding (M-LWR) problem. The difference with Kyber is that instead of explicitly adding error terms  $\mathbf{e}$ ,  $\mathbf{e}_1$ ,  $e_2$  from a “small” distribution, the errors are deterministically added by applying a rounding function mapping  $\mathbb{Z}_q$  to  $\mathbb{Z}_p$  with  $p < q$ . For Saber, both  $p$  and  $q$  are powers of two; therefore the rounding function is a shift extracting the  $\log_2(p)$  most significant bits of its input. We provide in Appendix G the description of a fully masked implementation of Saber, as we did for Kyber in the previous section.

## 8 Practical implementation and leakage evaluation

### 8.1 Implementation of Kyber and Saber

We have implemented in C a high-order version of the algorithm Kyber.Decaps, following the description of Section 6.2. For comparison, we have also performed a high-order implementation of the Saber algorithm. We refer to Appendix G.2 for a description. For both schemes, we have targeted the parameter set corresponding to NIST security category 3 (parameter Kyber768 for Kyber, see Table 7). We have run our implementation on a laptop and an embedded component. We provide the source code of the laptop implementation at:

[https://github.com/fragerar/HOTableConv/tree/main/Masked\\_KEMs](https://github.com/fragerar/HOTableConv/tree/main/Masked_KEMs)

For the embedded component, we have used a 100 MHz ARM Cortex-M3 architecture with 48k of RAM, which also includes a hardware accelerator for secure 32-bit random generation. Such component is used in real-life products like bank cards, passports, secure elements, etc.<sup>4</sup>

<sup>4</sup> Due to intellectual properties reasons, the component name or detailed description cannot be given.

The embedded code is almost the same as for the laptop implementation, except for the random generation which uses the hardware accelerator. We have also performed some RAM optimization in order to reduce the number of temporary variables without changing the number of operations.

**Kyber.** Our high-order implementation of `Kyber.Decaps` follows the description from Section 6.2. To generate random integers modulo  $q$ , we use the technique described in Algorithm 8 (see Appendix A), starting from a 32-bit random number generator. The timings are summarized in Table 12. For the embedded implementation, we can reach at most a security order of 3, due to RAM limitation.

	Security order $t$							
	0	1	2	3	4	5	6	7
Intel i7	133	1 164	2 225	4 723	6 613	11 177	14 174	19 806
ARM Cortex-M3	3 173	21 492	39 539	69 348	-	-	-	-

Table 12: `Kyber.Decaps` cycles counts on Intel(R) Core(TM) i7-1065G7 and ARM Cortex-M3, in thousands of cycles.

**Saber.** Our high-order implementation of `Saber.Decaps` follows the description from Appendix G.2. As for `Kyber`, the embedded implementation can reach at most a security order 3, due to RAM limitation. The timings are summarized in Table 13.

	Security order $t$							
	0	1	2	3	4	5	6	7
Intel i7	100	352	933	1 585	2 828	4 208	5 621	7 251
ARM Cortex-M3	5 682	17 805	42 662	67 389	-	-	-	-

Table 13: `Saber.Decaps` cycles counts on Intel(R) Core(TM) i7-1065G7, in thousands of cycles.

We see that for both `Kyber` and `Saber` the performance gap between the unmasked and the order 1 versions is fairly large. This is because we have used generic gadgets only, with no optimization at order 1. In practice, for first-order security, a significantly lower penalty factor could be obtained via some optimizations. In particular, all techniques based on table recomputation are much more efficient at order 1, since in that case the table can be randomized once and read multiple times.

## 8.2 Side-channel resistance evaluation

In order to validate our masking scheme beyond the theoretical framework, we have used the popular `ChipWhisperer-Lite` platform to get power traces of the execution on a Cortex-M4 core (`STM32F303`) of the basic zero-test gadgets from Section 3. The reason for performing benchmarks and experiments on different platforms (Intel, Cortex-M3, and Cortex-M4) is the following. Performing benchmarks on x86 enables to reach much higher security orders than for embedded implementations due to RAM limitation. For the embedded component, we have

used a Cortex-M3 component since it is widely used in real-life products (bank cards, passports, etc). For the side-channel evaluation we have used the ChipWhisperer-Lite which includes a Cortex-M4 by default.

For the leakage assessment on the ChipWhisperer-Lite platform, we have rewritten the gadgets specifically at order 1 in ARM assembly, in order to limit potential side-channel unsafe modifications from the compiler. We have conducted a fixed versus random  $t$ -test using the methodology described in [SM15]. The idea is to perform several measurement of the power consumption while the device under attack is executing the targeted gadget either with a fixed secret value chosen beforehand or with a random value sampled before each measurement. One keeps track of which measurement is using the fixed value, and thus one creates two sets of traces corresponding to the fixed and the random values respectively. The  $t$ -test will then be used as a distinguisher between the two sets at each point in the power traces. If the values output by the  $t$ -test are high, it means that the two sets seems statistically different and that an adversary can potentially use this information to learn something about the secret value. In practice, we have used a set of 10 000 traces for each gadget. For each trace, a coin was flipped to determine whether the random or the fixed secret value should be used.

The results of the  $t$ -test can be found in figures 1, 2 and 3. We see that when the RNG is switched off, the random and fixed inputs are distinguishable as the  $t$ -values are well above the usual threshold  $|t| > 4.5$ . When the random number generator is switched on, values are properly masked and the test is successful on all the zero-test gadgets.

In practice, obtaining these results was not trivial. Our first attempt was to directly take measurements using the C code used for the benchmarks in the previous section. Unfortunately, the traces were not leakage-free. We suspected that the compiler was generating assembly code that handled shares in an unsecured manner. To solve this issue, we decided to rewrite the gadgets directly in ARM assembly to obtain a full control over the instructions executed by the device. This greatly improved the situation, but in some cases, we still had some unexplained leakage due to the micro-architecture. More specifically, it appeared that manipulating the shares one after the other was a bad practice and would create leakage between instructions. This can be fixed by either rethinking the code in order to create some space or adding some dummy instructions.

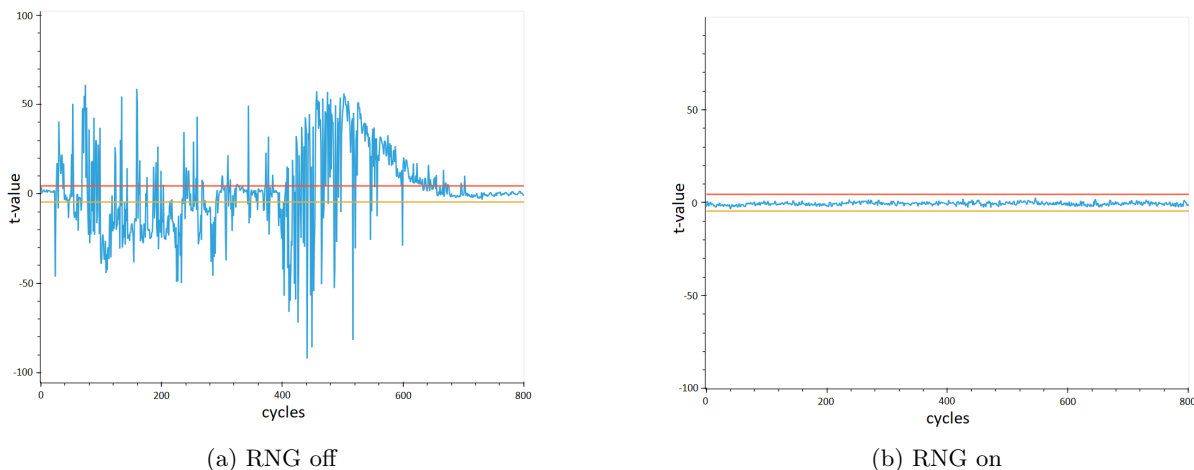


Figure 1:  $t$ -test ZT-Bool.



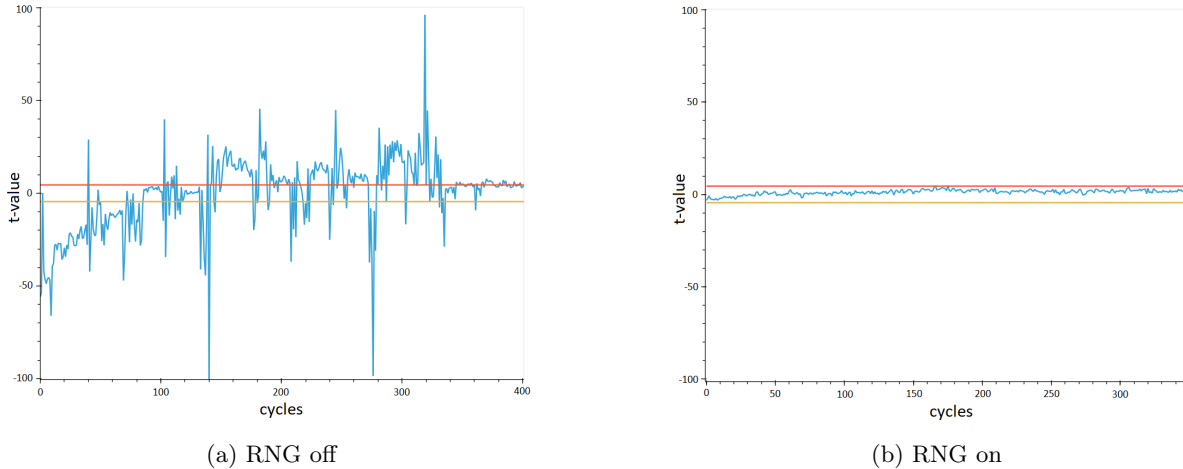


Figure 2:  $t$ -test ZT-Mul.

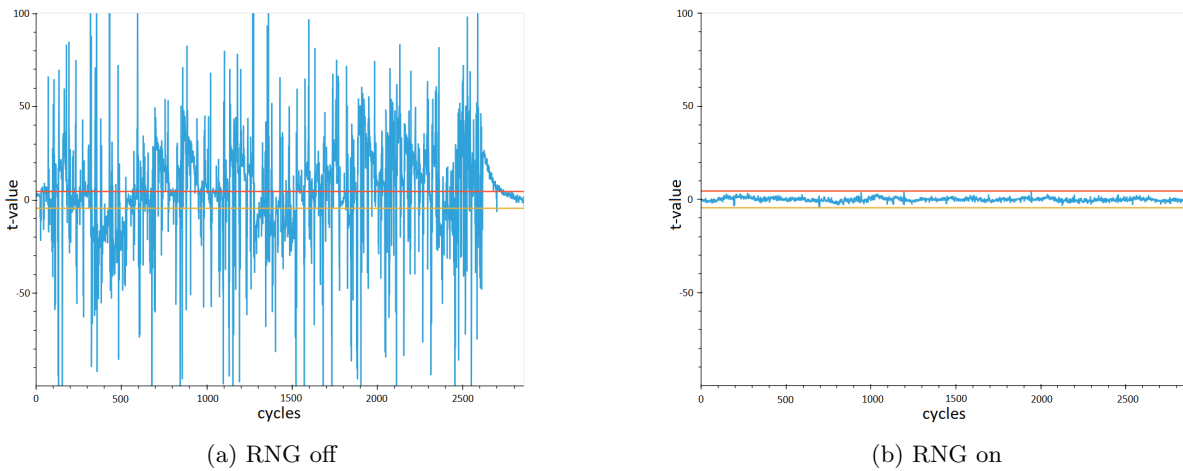


Figure 3:  $t$ -test ZT-Expo.

## 9 Conclusion

In this paper, we have described efficient techniques for high-order polynomial comparison, as used in lattice-based schemes with the Fujisaki-Okamoto transform. As an application, we have considered the high-order polynomial comparison in the NIST encryption standard Kyber. We have provided the first high-order description of the `Compress` function in Kyber, in order to perform the comparison on compressed ciphertexts. We have shown that the best approach is actually hybrid, with the `Compress` function being applied only on the last part of the ciphertext, while the rest is left uncompressed for the comparison. Finally, we have provided a complete description of the high-order masking of the IND-CCA decryption of the Kyber scheme at any order, with the practical results of a C implementation, and a  $t$ -test evaluation.

## References

- ABD<sup>+</sup>21. Roberto Avanzi, Joppe Bos, Léo Ducas, Eike Kiltz, Tancrede Lepoint, Vadim Lyubashevsky, John M. Schanck, Peter Schwabe, Gregor Seiler, and Damien Stehlé. CRYSTALS-Kyber (version 3.02) – submission to round 3 of the NIST post-quantum project. Specification document (update from August 2021). 2021-08-04., 2021.
- BBD<sup>+</sup>16. Gilles Barthe, Sonia Belaïd, François Dupressoir, Pierre-Alain Fouque, Benjamin Grégoire, Pierre-Yves Strub, and Rébecca Zucchini. Strong non-interference and type-directed higher-order masking. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*, pages 116–129, 2016. Publicly available at <https://eprint.iacr.org/2015/506.pdf>.
- BBE<sup>+</sup>18. Gilles Barthe, Sonia Belaïd, Thomas Espitau, Pierre-Alain Fouque, Benjamin Grégoire, Mélissa Rossi, and Mehdi Tibouchi. Masking the GLP lattice-based signature scheme at any order. In *Advances in Cryptology - EUROCRYPT 2018 - Proceedings, Part II*, pages 354–384, 2018.
- BC22. Olivier Bronchain and Gaëtan Cassiers. 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>.
- BDH<sup>+</sup>21. Shivam Bhasin, Jan-Pieter D’Anvers, Daniel Heinz, Thomas Pöppelmann, and Michiel Van Beirendonck. Attacking and defending masked polynomial comparison for lattice-based cryptography. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2021(3):334–359, 2021. <https://eprint.iacr.org/2021/104>.
- BDK<sup>+</sup>18. Joppe W. Bos, Léo Ducas, Eike Kiltz, Tancrede Lepoint, Vadim Lyubashevsky, John M. Schanck, Peter Schwabe, Gregor Seiler, and Damien Stehlé. CRYSTALS - kyber: A CCA-secure module-lattice-based KEM. In *2018 IEEE European Symposium on Security and Privacy, EuroS&P 2018, London, United Kingdom, April 24-26, 2018*, pages 353–367, 2018.
- BGR<sup>+</sup>21. Joppe W. Bos, Marc Gourjon, Joost Renes, Tobias Schneider, and Christine van Vredendaal. Masking Kyber: First- and higher-order implementations. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2021(4):173–214, 2021. <https://eprint.iacr.org/2021/483>.
- BMD<sup>+</sup>21. Andrea Basso, Jose Maria Bermudo Mera, Jan-Pieter D’Anvers, Angshuman Karmakar, Sujoy Sinha Roy, Michiel Van Beirendonck, and Frederik Vercauteren. Saber: Mod-LWR based KEM (round 3 submission), 2021. <https://www.esat.kuleuven.be/cosic/pqcrypto/saber/files/saberspecround3.pdf>.
- BPO<sup>+</sup>20. Florian Bache, Clara Paglialonga, Tobias Oder, Tobias Schneider, and Tim Güneysu. High-speed masking for polynomial comparison in lattice-based KEMs. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2020(3):483–507, 2020.
- CGMZ22. Jean-Sébastien Coron, François Gérard, Simon Montoya, and Rina Zeitoun. High-order table-based conversion algorithms and masking lattice-based encryption. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2022(2):1–40, 2022. <https://ia.cr/2021/1314>.
- CGTV15. Jean-Sébastien Coron, Johann Großschädl, Mehdi Tibouchi, and Praveen Kumar Vadnala. Conversion from arithmetic to boolean masking with logarithmic complexity. In *Proceedings of FSE 2015*, pages 130–149, 2015.
- CGV14. Jean-Sébastien Coron, Johann Großschädl, and Praveen Kumar Vadnala. Secure conversion between boolean and arithmetic masking of any order. In *Proceedings of CHES 2014*, pages 188–205, 2014.
- CJRR99. Suresh Chari, Charanjit S. Jutla, Josyula R. Rao, and Pankaj Rohatgi. Towards sound approaches to counteract power-analysis attacks. In *CRYPTO*, 1999.
- Cor14. Jean-Sébastien Coron. Higher order masking of look-up tables. In *Proceedings of EUROCRYPT 2014*, pages 441–458, 2014.
- CS21. Jean-Sébastien Coron and Lorenzo Spignoli. Secure shuffling in the probing model. *IACR Cryptol. ePrint Arch.*, 2021:258, 2021.
- DBV22. Jan-Pieter D’Anvers, Michiel Van Beirendonck, and Ingrid Verbauwhede. 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>.
- DHP<sup>+</sup>22. Jan-Pieter D’Anvers, Daniel Heinz, Peter Pessl, Michiel Van Beirendonck, and Ingrid Verbauwhede. Higher-order masked ciphertext comparison for lattice-based cryptography. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2022(2):115–139, 2022. <https://ia.cr/2021/1422>.
- FBR<sup>+</sup>21. Tim Fritzmann, Michiel Van Beirendonck, Debapriya Basu Roy, Patrick Karl, Thomas Schamberger, Ingrid Verbauwhede, and Georg Sigl. Masked accelerators and instruction set extensions for post-quantum cryptography. *IACR Cryptol. ePrint Arch.*, page 479, 2021.

- FO99. Eiichiro Fujisaki and Tatsuaki Okamoto. Secure integration of asymmetric and symmetric encryption schemes. In *CRYPTO '99, Proceedings*, pages 537–554, 1999.
- HCY20. Wei-Lun Huang, Jiun-Peng Chen, and Bo-Yin Yang. Power analysis on NTRU prime. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2020(1):123–151, 2020.
- HHK17. Dennis Hofheinz, Kathrin Hövelmanns, and Eike Kiltz. A modular analysis of the Fujisaki-Okamoto transformation. In *TCC 2017, Proceedings, Part I*, pages 341–371, 2017.
- ISW03. Yuval Ishai, Amit Sahai, and David A. Wagner. Private circuits: Securing hardware against probing attacks. In *CRYPTO 2003, Proceedings*, pages 463–481, 2003.
- LPR10. Vadim Lyubashevsky, Chris Peikert, and Oded Regev. On ideal lattices and learning with errors over rings. In *EUROCRYPT 2010. Proceedings*, pages 1–23, 2010.
- LS15. Adeline Langlois and Damien Stehlé. Worst-case to average-case reductions for module lattices. *Des. Codes Cryptogr.*, 75(3):565–599, 2015.
- Lum13. Jérémie O. Lumbroso. Optimal discrete uniform generation from coin flips, and applications. *CoRR*, abs/1304.1916, 2013.
- OSPG18. Tobias Oder, Tobias Schneider, Thomas Pöppelmann, and Tim Güneysu. Practical CCA2-secure and masked ring-lwe implementation. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2018(1):142–174, 2018.
- PPM17. Robert Primas, Peter Pessl, and Stefan Mangard. Single-trace side-channel attacks on masked lattice-based encryption. In *CHES 2017, Proceedings*, pages 513–533, 2017.
- RP10. Matthieu Rivain and Emmanuel Prouff. Provably secure higher-order masking of AES. In *CHES 2010, Proceedings*, pages 413–427, 2010.
- SM15. Tobias Schneider and Amir Moradi. Leakage assessment methodology - A clear roadmap for side-channel evaluations. In *CHES 2015. Proceedings*, volume 9293 of *Lecture Notes in Computer Science*, pages 495–513. Springer, 2015.
- SPOG19. Tobias Schneider, Clara Paglialonga, Tobias Oder, and Tim Güneysu. Efficiently masking binomial sampling at arbitrary orders for lattice-based crypto. In *PKC 2019, Proceedings, Part II*, pages 534–564, 2019.
- XPRO20. Zhuang Xu, Owen Pemberton, Sujoy Sinha Roy, and David F. Oswald. Magnifying side-channel leakage of lattice-based cryptosystems with chosen ciphertexts: The case study of Kyber. *IACR Cryptol. ePrint Arch.*, 2020:912, 2020.

## A Random generation modulo $q$

To generate a random integer in  $\mathbb{Z}_q$ , one can generate a random  $k$ -bit integer where the gap  $2^k - i \cdot q$  is small for some  $i$ . By rejection sampling, one obtains a uniformly distributed integer in  $[0, i \cdot q]$ , from which we obtain a uniformly distributed integer modulo  $q$ , with rejection probability  $1 - i \cdot q / 2^k$ . For example, with  $q = 3329$ , one can take  $k = 16$  and  $i = 19$ , with rejection probability 0.035.

We can also use the trick described in [Lum13, Section 3]. It consists in generating a random integer modulo  $q^2$ , which enables to extract two random integers modulo  $q$ ; we can of course use higher powers of  $q$ . As previously, we generate a random  $k$ -bit integer such that the gap  $2^k - i \cdot q^2$  is small. The rejection probability is then  $1 - i \cdot q^2 / 2^k$ . For example, with  $q = 3329$ , we can use  $k = 25$  and  $i = 3$ , and the rejection probability is 0.009, so we are using 12.5 bits per random integer modulo  $q$ , with rejection probability 0.0046 per random integer. We can also use  $k = 32$  and  $i = 387$ , which gives 16 bits per random integer as previously, but with rejection probability 0.0007 per random integer (instead of 0.035, so a factor 50 improvement in rejection rate). We describe the pseudo-code below, to be run with parameters  $(i, k, q) = (387, 32, 3329)$ .

---

**Algorithm 8** randomModq

---

**Input:** Parameters  $i, k, q$  such that  $i \cdot q^2 < 2^k$ .

**Output:**  $r_1, r_2$  uniformly distributed in  $\mathbb{Z}_q$ .

```
1:  $r := 2^k$ 
2: while  $r \geq i \cdot q^2$  do
3:    $r \leftarrow \{0, 1\}^k$ 
4: end while
5:  $r := r \bmod q^2$ 
6: return  $(r \bmod q, \lfloor r/q \rfloor)$ 
```

---

## B Share recombination

### B.1 Share recombination with $n$ linear mask refreshing

When performing the comparison between two ciphertexts  $c$  and  $c'$ , the output bit  $b$  of the comparison must eventually be computed in the clear, which means that the  $n$  shares  $b_i$  of  $b$  must eventually be recombined. We first recall the approach used in [Cor14] for recombining the shares when outputting the AES ciphertext. It consists in performing a sequence of  $n$  mask refreshing, each of complexity  $\mathcal{O}(n)$ , so that the share recombination can be perfectly simulated, knowing the output bit  $b$ . Namely, this output bit  $b$  of the comparison can be given for free to the simulator, since that bit  $b$  is eventually known by the adversary. Following [Cor14], this enables to prove the  $(n - 1)$ -NIo property of the share recombination algorithm, according to Definition 3 from [BBE<sup>+</sup>18]. In Appendix B.2, we describe a slightly more efficient approach, still with complexity  $\mathcal{O}(n^2)$ , but using only half the randomness.

We first recall the LinearRefreshMasks algorithm from [RP10], working in any finite field  $\mathbb{F}$ .

---

**Algorithm 9** LinearRefreshMasks

---

**Input:**  $x_1, \dots, x_n \in \mathbb{F}$

**Output:**  $y_1, \dots, y_n \in \mathbb{F}$  such that  $y_1 + \dots + y_n = x_1 + \dots + x_n$

```
1:  $y_n \leftarrow x_n$ 
2: for  $j = 1$  to  $n - 1$  do
3:    $r_j \leftarrow \mathbb{F}$ 
4:    $y_j \leftarrow x_j + r_j \bmod q$ 
5:    $y_n \leftarrow y_n - r_j \bmod q$ 
6: end for
7: return  $y_1, \dots, y_n$ 
```

---

---

**Algorithm 10** RecombineShares [Cor14]

---

**Input:**  $y_1, \dots, y_n$ **Output:**  $y$  such that  $y = y_1 + \dots + y_n$ 1: **for**  $i = 1$  **to**  $n$  **do**  $(y_1, \dots, y_n) \leftarrow \text{LinearRefreshMasks}(y_1, \dots, y_n)$ 2:  $c \leftarrow y_1$ 3: **for**  $i = 2$  **to**  $n$  **do**  $c \leftarrow c + y_i$ 4: **return**  $c$ 

---

Following the same approach as in [Cor14], we prove the following lemma.

**Lemma 2.** *The RecombineShares algorithm is  $(n - 1)$ -Nlo, when  $y$  is given to the simulator.*

*Proof.* Since the adversary has at most  $t < n$  probes, at least one of the LinearRefreshMasks has not been probed. Let  $1 \leq i^* \leq n$  be the corresponding index. Any probe for  $i < i^*$  can be simulated by including some index  $j$  in a set  $I$ , initially empty. Given the knowledge of  $y$ , the  $n$  outputs of the LinearRefreshMasks at index  $i^*$  can be perfectly simulated, simply by generating random  $y_1, \dots, y_n$  such that  $y = y_1 + \dots + y_n$ . Therefore, any probe after the index  $i^*$  can be perfectly simulated. Finally, any probe can be perfectly simulated from the knowledge of  $y$  and the input  $y_{|I}$ , with  $|I| \leq t$ .  $\square$

## B.2 Share recombination: a more efficient approach

In the following, we describe a variant of the above share recombination technique, using only half the randomness. For this, we use the following extension of the  $t$ -SNI security notion recently introduced in [CS21], which was used to prove the security of the ISW construction in the stateful model. Under this definition called free-SNI, all output variables except one can always be perfectly simulated (which is not necessarily the case in the original SNI definition). Moreover it was shown in [CS21] that the RefreshMasks algorithm (which we recall in Appendix C.2) satisfies the extended notion.

**Definition 4 (Free- $t$ -SNI security).** *Let  $G$  be a gadget taking as input  $n$  shares  $(a_i)_{1 \leq i \leq n}$  and outputting  $n$  shares  $(b_i)_{1 \leq i \leq n}$ . The gadget  $G$  is said to be free  $t$ -SNI secure if for any set of  $t_1 \leq t$  probed intermediate variables, there exists a subset  $I$  of input indices with  $|I| \leq t_1$ , such that the  $t_1$  intermediate variables and the output variables  $b_{|I}$  can be perfectly simulated from  $a_{|I}$ , while for any  $O \subsetneq [1, n] \setminus I$  the output variables in  $b_{|O}$  are uniformly and independently distributed, conditioned on the probed variables and  $b_{|I}$ .*

**Lemma 3 ([CS21]).** *The RefreshMasks algorithm is free- $(n - 1)$ -SNI.*

Thanks to the free-SNI definition, we can now simulate all output variables, if the simulator is given the value encoded by those output variables, that is  $b = b_1 + \dots + b_n$ . We can then recombine the output shares of the gadget, and all intermediate variables in the recombination can be perfectly simulated.

In particular, assume that we must recombine the shares  $a_1, \dots, a_n$ . For this, we first compute

$$(b_1, \dots, b_n) \leftarrow \text{RefreshMasks}(a_1, \dots, a_n)$$

and eventually compute  $b = b_1 + \dots + b_n$ , which gives  $b = a_1 + \dots + a_n$  as required. The advantage of using RefreshMasks instead of  $n$  LinearRefreshMasks as in the previous section, is that we use  $n(n - 1)/2$  random values instead of  $n(n - 1)$ .

**Lemma 4.** *Let  $G$  be a gadget taking as input  $n$  shares  $(a_i)_{1 \leq i \leq n}$  and outputting  $n$  shares  $(b_i)_{1 \leq i \leq n}$ . Assume that  $G$  satisfies the free- $t$ -SNI property. Then for any set of  $t_1 \leq t$  intermediate variables, there exists a subset  $I$  of input indices with  $|I| \leq t_1$ , such that the  $t_1$  intermediate variables and all output variables  $(b_i)_{1 \leq i \leq n}$  can be perfectly simulated from  $a_{|I}$  and  $b = b_1 + \dots + b_n$ .*

*Proof.* We use the set  $I$  obtained from Definition 4. If  $|I| = n$ , we can simulate all output variables. Otherwise, let  $i^* \notin I$ . We let  $O$  such that  $I \cup O = [1, n] \setminus \{i^*\}$ . From the free- $t$ -SNI definition, we can simulate all variables in  $b_{I \cup O}$ . The remaining variable  $b_{i^*}$  is simulated from the knowledge of  $b$  as  $b_{i^*} = b - \sum_{i \neq i^*} b_i$ .  $\square$

From Lemma 4, all output variables of a free-SNI gadget can be simulated using the knowledge of the encoded value  $b$ . Therefore, all intermediate variables in the subsequent recombination can be perfectly simulated. This shows that the resulting gadget satisfies the NI property.<sup>5</sup> This enables to prove the probing security of the full circuit by composition.

**Corollary 1 (NI security).** *Let  $G$  be a gadget taking as input a sequence of  $n$  shares  $(a_i)_{1 \leq i \leq n}$  and outputting  $n$  shares  $(b_i)_{1 \leq i \leq n}$ , and let  $G'$  be the same as  $G$  but outputting  $b = b_1 + \dots + b_n$ . Assume that  $G$  satisfies the free- $t$ -SNI property. Then for any set of  $t_1 \leq t$  intermediate variables of gadget  $G'$ , there exists a subset  $I$  of input indices with  $|I| \leq t_1$ , such that the  $t_1$  intermediate variables can be perfectly simulated from  $a_{|I}$  and  $b$ .*

We stress that in Lemma 4, even if  $t_1 = n - 1$  of the output variables are probed and the output bit  $b$  is known, only  $t_1 = n - 1$  input variables  $a_i$  of the gadget must be known, which means that only  $n - 1$  output variables of the preceding gadget are required for the simulation. Whereas, without a free-SNI gadget such as RefreshMasks, all  $n$  input shares must be known for the recombination, which prevents to prove the probing security of the full circuit.

## C Zero testing a single value

### C.1 The SecAnd algorithm

We first recall the SecAnd algorithm that enables to compute the And between two Boolean masked values with  $n$  shares. The algorithm is a variant with  $k$ -bit words of the original ISW algorithm. The algorithm has complexity  $\mathcal{O}(n^2)$ , with a number of operations :

$$T_{\text{SecAnd}}(n) = n(7n - 5)/2$$

**Lemma 5 ([BBD<sup>+</sup>16]).** *The SecAnd algorithm is  $(n - 1)$ -SNI.*

### C.2 Mask refreshing

We recall the RefreshMasks algorithm, where the operations are performed in any group  $\mathbb{G}$ , for example the additive group  $\mathbb{Z}_q$  for any integer  $q$ .

<sup>5</sup> The SNI property would not really be applicable in that case, since by assumption the gadget outputs a single bit  $b$  and not a  $n$ -sharing of  $b$ .

---

**Algorithm 11** SecAnd

---

**Input:**  $k \in \mathbb{N}$ ,  $x_1, \dots, x_n \in \{0, 1\}^k$ ,  $y_1, \dots, y_n \in \{0, 1\}^k$ **Output:**  $z_1, \dots, z_n \in \{0, 1\}^k$ , with  $\bigoplus_{i=1}^n z_i = (\bigoplus_{i=1}^n x_i) \wedge (\bigoplus_{i=1}^n y_i)$ 

```
1: for  $i = 1$  to  $n$  do  $z_i \leftarrow x_i \wedge y_i$ 
2: for  $i = 1$  to  $n$  do
3:   for  $j = i + 1$  to  $n$  do
4:      $r \leftarrow \{0, 1\}^k$ 
5:      $r' \leftarrow (r \oplus (x_i \wedge y_j)) \oplus (x_j \wedge y_i)$ 
6:      $z_i \leftarrow z_i \oplus r$ 
7:      $z_j \leftarrow z_j \oplus r'$ 
8:   end for
9: end for
10: return  $z_1, \dots, z_n$ 
```

---

---

**Algorithm 12** RefreshMasks

---

**Input:**  $a_1, \dots, a_n$ **Output:**  $c_1, \dots, c_n$  such that  $\sum_{i=1}^n c_i = \sum_{i=1}^n a_i$ 

```
1: For  $i = 1$  to  $n$  do  $c_i \leftarrow a_i$ 
2: for  $i = 1$  to  $n$  do
3:   for  $j = i + 1$  to  $n$  do
4:      $r \leftarrow \mathbb{G}$ ,  $c_i \leftarrow c_i + r$ ,  $c_j \leftarrow c_j - r$ 
5:   end for
6: end for
7: return  $c_1, \dots, c_n$ 
```

---

The algorithm has complexity  $\mathcal{O}(n^2)$ , with a number of operations

$$T_{\text{refresh}}(n) = 3n(n - 1)/2$$

**Lemma 6** ([BBD<sup>+</sup>16]). *The RefreshMasks algorithm is  $(n - 1)$ -SNI.*

### C.3 Boolean zero testing in $\{0, 1\}^k$

We consider the zero-testing of  $x \in \{0, 1\}^k$  from its Boolean shares. We consider the  $k$  bits of  $x = x^{(k-1)} \dots x^{(0)}$ . The zero-testing of  $x$  computes a bit  $b$  with  $b = 1$  if  $x = 0$ , and  $b = 0$  otherwise; therefore:

$$b = \overline{x^{(k-1)} \vee \dots \vee x^{(0)}} = \overline{x^{(k-1)}} \wedge \dots \wedge \overline{x^{(0)}}$$

Starting from the  $n$  Boolean shares of  $x = x_1 \oplus \dots \oplus x_n$ , the right-hand side of the above equation can be computed by a sequence of  $k - 1$  secure **And** (see Appendix C.1). For simplicity we actually perform  $k$  iterations of **SecAnd**, the first one being a **SecAnd** with encoded input 1, to avoid an explicit mask refreshing at the beginning. The shares  $b_1, \dots, b_n$  are eventually recombined using the **RecombineShares** algorithm (see Appendix B.1). We obtain Algorithm 13 below.

---

**Algorithm 13** ZeroTestBool

---

**Input:**  $k \in \mathbb{N}$  and  $x_1, \dots, x_n \in \{0, 1\}^k$

**Output:**  $b \in \{0, 1\}$  such that  $b = 1$  if  $\oplus x_i = 0$ , and  $b = 0$  otherwise.

1:  $(y_1, \dots, y_n) \leftarrow (\bar{x}_1, x_2, \dots, x_n)$

2:  $(b_1, \dots, b_n) \leftarrow (1, 0, \dots, 0)$

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

4:      $(b_1, \dots, b_n) \leftarrow \text{SecAnd}(1, (b_1, \dots, b_n), ((y_1 \gg j) \& 1, \dots, (y_n \gg j) \& 1))$

5: **end for**

6: **return**  $\text{RecombineShares}(b_1, \dots, b_n)$ 

---

**Theorem 8.** *The ZeroTestBool is  $(n - 1)$ -NI.*

*Proof.* The ZeroTestBool algorithm up to Line 5 is  $(n - 1)$ -SNI since it is the composition of SecAnd operations, which are  $(n - 1)$ -SNI. Thanks to the final RecombineShares and Lemma 2, the ZeroTestBool algorithm is  $(n - 1)$ -NI, when  $b$  is given to the simulator.

Alternatively, at Line 6 one can replace the RecombineShares algorithm by the RefreshMasks algorithm, followed by the computation of  $b = b_1 \oplus \dots \oplus b_n$ . In that case, the ZeroTestBool algorithm up to RefreshMasks is free- $(n - 1)$ -SNI. From Corollary 1, this implies that the full ZeroTestBool is  $(n - 1)$ -NI, when  $b$  is given to the simulator.  $\square$

The advantage of using RefreshMasks instead of RecombineShares is that we use  $n(n - 1)/2$  random values instead of  $n(n - 1)$ . Therefore, in the rest of the paper, we will use this later approach.

#### C.4 Boolean zero-test with complexity $\mathcal{O}(n^2 \cdot \log k)$

For a  $k$ -bit input  $x$ , the previous algorithm has complexity  $\mathcal{O}(n^2 \cdot k)$ . In the following, we describe an improved algorithm ZeroTestBoolLog with complexity  $\mathcal{O}(n^2 \cdot \log k)$ , by taking advantage of the And operations on  $k$ -bit registers, instead of single bits. We also provide a more precise operation count.

Let  $x \in \{0, 1\}^k$  and let  $x = x_1 \oplus \dots \oplus x_n$  a Boolean sharing of  $x$ . We describe a procedure to zero-test  $x$  in  $\mathcal{O}(n^2 \cdot \log k)$  operations on  $k$ -bit registers, instead of  $\mathcal{O}(n^2 \cdot k)$  with the previous approach. The technique is as follows. We write

$$x = x^{(k-1)} \dots x^{(0)}$$

the  $k$  bits of  $x$ . Let  $m = \lceil \log_2 k \rceil$ . If  $k$  is not a power of two, then we set the most significant bits of  $x$  to 1 until the next power of two, which is  $2^m$ . Let  $f_i(x) = x \wedge (x \gg 2^i)$ . We prove below that we have:

$$x^{(k-1)} \wedge \dots \wedge x^{(0)} = \text{LSB}((f_{m-1} \circ \dots \circ f_0)(x)) \quad (7)$$

Therefore to zero-test  $x$ , we can compute:

$$\overline{x^{(k-1)} \vee \dots \vee x^{(0)}} = \text{LSB}((f_{m-1} \circ \dots \circ f_0)(\bar{x}))$$

We describe in Algorithm 14 below the high-order computation of the previous equation with  $n$  shares, using the SecAnd and RefreshMasks algorithms from sections C.1 and C.2.



---

**Algorithm 14** ZeroTestBoolLog

---

**Input:**  $k \in \mathbb{Z}$  and  $x_1, \dots, x_n \in \{0, 1\}^k$

**Output:**  $b \in \{0, 1\}$  with  $b = 1$  if  $\bigoplus_{i=1}^n x_i = 0$  and  $b = 0$  otherwise

```
1:  $m \leftarrow \lceil \log_2 k \rceil$ 
2:  $y_1 \leftarrow \overline{x_1}$  or  $(2^{2^m} - 2^k)$ 
3: for  $i = 2$  to  $n$  do  $y_i \leftarrow x_i$ 
4: for  $i = 0$  to  $m - 1$  do
5:    $(z_1, \dots, z_n) \leftarrow \text{RefreshMasks}(y_1 \gg 2^i, \dots, y_n \gg 2^i)$ 
6:    $(y_1, \dots, y_n) \leftarrow \text{SecAnd}(m, (y_1, \dots, y_n), (z_1, \dots, z_n))$ 
7: end for
8: return  $\text{RecombineShares}(y_1 \& 1, \dots, y_n \& 1)$ 
```

---

**Theorem 9 (Soundness).** *Given as input  $x_1, \dots, x_n \in \{0, 1\}^k$ , the algorithm ZeroTestBoolLog outputs  $b = 1$  if  $\bigoplus_{i=1}^n x_i = 0$  and  $b = 0$  otherwise.*

*Proof.* We first consider the case where  $n = 1$ , that is  $x = x_1$  and  $y = y_1 = \overline{x}$ . For simplicity, we first assume that  $k$  is a power of two, that is  $k = 2^m$ . At Step 6 of the ZeroTestBoolLog algorithm, we compute  $(f_{m-1} \circ \dots \circ f_0)(y)$  where  $f_i(y) = y \wedge (y \gg 2^i)$ . To ease reading, we denote by  $F_i(y)$  the value  $(f_i \circ \dots \circ f_0)(y)$ . In the following we prove by induction on  $i$  for  $i < m$  that the  $j$ -th bit of  $F_i(y)$  is

$$(F_i(y))^{(j)} = y^{(j+2^{i+1}-1)} \wedge \dots \wedge y^{(j)}, \quad (8)$$

for  $j \leq 2^m - 2^{i+1}$ . For the base case  $i = 0$ , we have

$$(F_0(y))^{(j)} = (f_0(y))^{(j)} = ((y \gg 1) \wedge y)^{(j)} = (y \gg 1)^{(j)} \wedge y^{(j)} = y^{(j+1)} \wedge y^{(j)}$$

which satisfies the recurrence hypothesis. Now we show that

$$(F_{i+1}(y))^{(j)} = y^{(j+2^{i+2}-1)} \wedge \dots \wedge y^{(j)}.$$

Indeed, we have

$$\begin{aligned} (F_{i+1}(y))^{(j)} &= (f_{i+1}(F_i(y)))^{(j)} = (F_i(y) \wedge ((F_i(y) \gg 2^{i+1})))^{(j)} \\ &= (F_i(y))^{(j)} \wedge (F_i(y))^{(j+2^{i+1})} \end{aligned}$$

By using the recurrence hypothesis in Equation (8), we get

$$\begin{aligned} (F_{i+1}(y))^{(j)} &= (y^{(j+2^{i+1}-1)} \wedge \dots \wedge y^{(j)}) \wedge (y^{((j+2^{i+1})+2^{i+1}-1)} \wedge \dots \wedge y^{(j+2^{i+1})}) \\ &= y^{(j+2^{i+2}-1)} \wedge \dots \wedge y^{(j)} \end{aligned}$$

which terminates the recursive proof.

In particular, for  $i = m - 1$ , we can use Equation (8) for  $j \leq 2^m - 2^{i+1} = 0$ . Thus, by keeping only the LSB part (that is  $j = 0$ ) as done in Step 8 of Algorithm ZeroTestBoolLog, we have:

$$\begin{aligned} (F_{m-1}(y))^{(0)} &= \text{LSB}((f_{m-1} \circ \dots \circ f_0)(y)) \\ &= y^{(2^m-1)} \wedge \dots \wedge y^{(0)} = y^{(k-1)} \wedge \dots \wedge y^{(0)}, \end{aligned}$$

as specified in equation (7). Note that this is also true in the case where  $k$  is not a power of two since in this case, the most significant bits of  $y$  are initially set to 1. From  $y = \bar{x}$ , we obtain as required:

$$\text{LSB}((f_{m-1} \circ \dots \circ f_0)(\bar{x})) = \overline{y^{(k-1)} \vee \dots \vee y^{(0)}} = \overline{x^{(k-1)} \vee \dots \vee x^{(0)}}.$$

Eventually, the result also holds for  $x = x_1 \oplus \dots \oplus x_n$  with  $n > 1$ , since the same operations are performed on all shares, which proves the theorem.  $\square$

**Complexity.** We have using  $T_{\text{refresh}}(n) = 3n(n-1)/2$  and  $T_{\text{SecAnd}}(n) = n(7n-5)/2$ :

$$\begin{aligned} T_{\text{ZeroTestBoolLog}}(k, n) &= 2 + \lceil \log_2 k \rceil \cdot (n + T_{\text{refresh}}(n) + T_{\text{SecAnd}}(n)) \\ &\quad + n + T_{\text{refresh}}(n) + n - 1 \\ &\simeq 5n^2 \lceil \log_2 k \rceil \end{aligned}$$

**Theorem 10.** *The ZeroTestBoolLog algorithm is  $(n-1)$ -Nlo.*

*Proof.* It is easy to see that the composition of steps 5 and 6 is  $(n-1)$ -SNI secure, from the  $(n-1)$ -SNI security of SecAnd and RefreshMasks. Therefore, the ZeroTestBoolLog algorithm up to Line 7 satisfies the  $(n-1)$ -SNI property, and therefore the full algorithm is  $(n-1)$ -Nlo when  $b$  is given to the simulator.  $\square$

### C.5 Zero testing modulo $q$ via arithmetic to Boolean conversion

We now consider the zero-testing of an element  $x \in \mathbb{Z}_q$  from its arithmetic shares. Given as input the  $n$  arithmetic shares of  $x = x_1 + \dots + x_n \pmod q$ , we must output a bit  $b$ , with  $b = 1$  if  $x = 0$  and  $b = 0$  if  $x \neq 0$ , without revealing more information about  $x$ . For  $q \leq 2^k$ , we first perform an arithmetic to Boolean conversion, which gives the Boolean shares  $y_1, \dots, y_n \in \{0, 1\}^k$ , with  $x = y_1 \oplus \dots \oplus y_n$ . We then apply the Boolean zero-testing algorithm from the previous section. We obtain the pseudo-code below.

---

#### Algorithm 15 ZeroTestAB

---

**Input:**  $q \in \mathbb{Z}$ ,  $k \in \mathbb{Z}$  with  $q \leq 2^k$ , and  $x_1, \dots, x_n \in \mathbb{Z}_q$

**Output:**  $b \in \{0, 1\}$  with  $b = 1$  if  $\sum_i x_i = 0 \pmod q$  and  $b = 0$  otherwise

1:  $(y_1, \dots, y_n) \leftarrow \text{ArithmeticToBoolean}(q, (x_1, \dots, x_n))$

2: **return**  $\text{ZeroTestBoolLog}(k, (y_1, \dots, y_n))$

---

The arithmetic to Boolean conversion step has complexity  $\mathcal{O}(n^2 \cdot k)$  for  $q = 2^k$ , using [CGV14] or the table recomputation approach from [CGMZ22]. The technique actually works for arithmetic masking modulo any integer  $q$ , since we can use [BBE<sup>+</sup>18, SPOG19] to convert from arithmetic modulo  $q$  to Boolean masking, with complexity  $\mathcal{O}(n^2 \cdot k)$  for a  $k$ -bit modulus  $q$ . In the second step, one can use the improved algorithm ZeroTestBoolLog from Appendix C.4 with complexity  $\mathcal{O}(n^2 \cdot \log k)$ . Therefore the overall complexity is  $\mathcal{O}(n^2 \cdot k)$ , where  $k = \lceil \log_2 q \rceil$ , with a number of operations:

$$T_{\text{ZeroTestAB}}(k, n) = T_{\text{AB}}(k, n) + T_{\text{ZeroTestBoolLog}}(k, n)$$

where  $T_{\text{AB}}(k, n)$  is the complexity of the arithmetic to Boolean conversion for a  $k$ -bit modulus  $q$ .

**Theorem 11.** *The ZeroTestAB algorithm is  $(n - 1)$ -NI.*

*Proof.* The result follows from Theorem 8, with the ArithmeticToBoolean algorithm which is assumed to be  $(n - 1)$ -NI.  $\square$

### C.6 Zero testing modulo $2^k$ for small $k$ via table recomputation

The technique is a direct application of the table-based conversion algorithm from [CGMZ22]. Namely the table recomputation from [CGMZ22] can high-order compute any function  $f : G \rightarrow H$ , for any groups  $G$  and  $H$ . So it suffices to take  $G = \mathbb{Z}_q$  and  $H = \{0, 1\}$ , with  $f(x) = 1$  if  $x = 0 \pmod{q}$  and  $f(x) = 0$  otherwise. The technique has complexity  $\mathcal{O}(q \cdot n^2)$ , which can be prohibitive for large  $q$ . For  $q = 2^k$  and small  $k$ , we can use the register optimization from [CGMZ22]. In that case the countermeasure has complexity  $\mathcal{O}(n^2)$  only, assuming that we have access to  $2^k$ -bit registers. Therefore this optimization can only work for small  $k$ , say up to  $k = 8$ .

More precisely, the technique first initializes a table  $T$  with  $q$  rows, where for  $0 \leq i < q$  the  $i$ -th row contains a  $n$ -shared Boolean encoding of 1 for  $i = 0$ , and 0 otherwise:

$$\begin{aligned} T(0) &= (1, 0, \dots, 0) \\ T(1) &= (0, 0, \dots, 0) \\ &\vdots \\ T(q - 1) &= (0, 0, \dots, 0) \end{aligned}$$

Given as input the shares  $x_i$  with  $x = x_1 + \dots + x_n \pmod{q}$ , one progressively shifts the rows of the table by successive shares  $x_i$ , up to the  $x_{n-1}$  share. The encodings are refreshed between each successive shift. Eventually the table has been shifted by  $x_1 + \dots + x_{n-1} \pmod{q}$ , so it suffices to read the table at row  $x_n$  to obtain an encoding of 1 if  $x = 0$ , and 0 otherwise. The register optimization consists in putting each column of the table in a register, so that the shifting of the table by each  $x_i$  corresponds to a rotation by  $x_i$  of each of the  $n$  registers, which is much faster. We provide the pseudocode below. We denote by  $R_j[u]$  the  $u$ -th bit of register  $R_j$  and we denote by  $\text{ROR}[a](R)$  the cyclic rotation of a register  $R$  by  $a$  bits to the right.

---

**Algorithm 16** ZeroTestTable

---

**Input:**  $x_1, \dots, x_n \in \mathbb{Z}_{2^k}$ **Output:**  $b \in \{0, 1\}$ , with  $b = 1$  if  $x_1 + \dots + x_n = 0 \pmod{2^k}$ , and 0 otherwise.

```
1:  $R_1[0] \leftarrow 1$ 
2: for all  $1 \leq u < 2^k$  do  $R_1[u] \leftarrow 0$ 
3: for all  $2 \leq j \leq n$  do  $R_j \leftarrow 0$ .
4: for  $i = 1$  to  $n - 1$  do
5:   for  $j = 1$  to  $n$  do  $R_j \leftarrow \text{ROR}[x_i](R_j)$ 
6:   for  $j = 1$  to  $n - 1$  do
7:      $r \leftarrow \{0, 1\}^{2^k}$ ,  $R_j \leftarrow R_j \oplus r$ ,  $R_n \leftarrow R_n \oplus r$ 
8:   end for
9: end for
10:  $(b_1, \dots, b_n) \leftarrow \text{RefreshMasks}_{\{0,1\}}(R_1[x_n], \dots, R_n[x_n])$ 
11: return  $b_1 \oplus \dots \oplus b_n$ 
```

---

**Theorem 12.** *The ZeroTestTable algorithm is  $(n - 1)$ -Nlo.*

*Proof.* From [CGMZ22], the ZeroTestTable algorithm up to Line 10 is  $(n - 1) - \text{SNI}$ . Thanks to the last RefreshMasks, it is actually free- $(n - 1) - \text{SNI}$ . Therefore the full algorithm is  $(n - 1) - \text{Nlo}$  when the output  $b$  is given to the simulator.  $\square$

## C.7 Secure Multiplication modulo $q$

We recall hereafter the SecMult algorithm as already considered in [SPOG19]. Note that the number of operations of SecMult is  $n \cdot (7n - 5)/2$  by considering random generation in  $\mathbb{Z}_q$ , addition and multiplication modulo  $q$  as a single operation.

---

**Algorithm 17** SecMult

---

**Input:**  $x_1, \dots, x_n \in \mathbb{Z}_q$ ,  $y_1, \dots, y_n \in \mathbb{Z}_q$ **Output:**  $z_1, \dots, z_n \in \mathbb{Z}_q$  such that  $\sum_i z_i = (\sum_i x_i) \cdot (\sum_i y_i) \pmod{q}$ .

```
1: for  $i = 1$  to  $n$  do  $z_i \leftarrow x_i \cdot y_i \pmod{q}$ 
2: for  $i = 1$  to  $n$  do
3:   for  $j = i + 1$  to  $n$  do
4:      $r \leftarrow \mathbb{Z}_q$ 
5:      $r' \leftarrow (r + x_i \cdot y_j \pmod{q}) + x_j \cdot y_i \pmod{q}$ 
6:      $z_i \leftarrow z_i - r \pmod{q}$ 
7:      $z_j \leftarrow z_j + r' \pmod{q}$ 
8:   end for
9: end for
```

---

## C.8 Zero testing modulo a prime $q$ via exponentiation

---

### Algorithm 18 SecExpo

---

**Input:**  $A_1, \dots, A_n \in \mathbb{Z}_q$  with  $\sum_i A_i = x \pmod{q}$  for prime  $q$ , an exponent  $e$ .

**Output:**  $B_1, \dots, B_n \in \mathbb{Z}_q$  with  $\sum_i B_i = x^e \pmod{q}$

```

1:  $(B_1, \dots, B_n) \leftarrow (1, 0, \dots, 0)$ 
2: for  $i = \lceil \log_2 e \rceil$  to 0 do
3:    $(C_1, \dots, C_n) \leftarrow \text{RefreshMasks}(B_1, \dots, B_n)$ 
4:    $(B_1, \dots, B_n) \leftarrow \text{SecMult}((B_1, \dots, B_n), (C_1, \dots, C_n))$ 
5:   if  $(e \& 2^i) = 2^i$  then  $(B_1, \dots, B_n) \leftarrow \text{SecMult}((B_1, \dots, B_n), (A_1, \dots, A_n))$ 
6: end for
7: return  $(B_1, \dots, B_n)$ 

```

---



---

### Algorithm 19 ZeroTestExpoShares

---

**Input:**  $A_1, \dots, A_n \in \mathbb{Z}_q$  for prime  $q$ .

**Output:**  $B_1, \dots, B_n$  with  $\sum_i B_i = 1 \pmod{q}$  if  $\sum_i A_i = 0 \pmod{q}$  and  $\sum_i B_i = 0 \pmod{q}$  otherwise

```

1:  $(B_1, \dots, B_n) \leftarrow \text{SecExpo}((A_1, \dots, A_n), q - 1)$ 
2:  $(B_1, \dots, B_n) \leftarrow (1 - B_1 \pmod{q}, -B_2 \pmod{q}, \dots, -B_n \pmod{q})$ 
3: return  $(B_1, \dots, B_n)$ 

```

---



---

### Algorithm 20 ZeroTestExpo

---

**Input:**  $A_1, \dots, A_n \in \mathbb{Z}_q$  for prime  $q$ .

**Output:**  $b \in \{0, 1\}$  with  $b = 1$  if  $\sum_i A_i = 0 \pmod{q}$  and  $b = 0$  otherwise

```

1:  $(B_1, \dots, B_n) \leftarrow \text{ZeroTestExpoShares}(A_1, \dots, A_n)$ 
2: return  $\text{RecombineShares}(B_1, \dots, B_n)$ 

```

---

**Complexity.** The complexity of the SecExpo algorithm with  $e = q - 1$  is  $\mathcal{O}(n^2 \cdot \log q)$ , assuming that a multiplication modulo  $q$  takes unit time. More precisely, the number of operations of SecMult is  $n \cdot (7n - 5)/2$ . The number of operation of RefreshMasks is  $n \cdot (3n - 1)/2$ . For  $q = 3329$ , the algorithm requires 13 squares and 4 multiplies. This means 4 RefreshMasks and 17 SecMult. The total number of operations for SecExpo is therefore  $n \cdot (131n - 89)/2$ . Eventually, the number of operations for ZeroTestExpoShares is:

$$T_{\text{ZeroTestExpoShares}}(n) = n \cdot (131n - 87)/2$$

and is finally  $n \cdot (67n - 43)$  for ZeroTestExpo.

### C.9 Proof of Theorem 1

We describe hereafter the construction of the set  $I \subset [1, n]$  of indices. Initially,  $I$  is empty. For every probed input variable  $x_i$  and for any probed intermediate variable  $B_i$  at Loop  $j$  between Steps 3 and 5, for  $1 \leq i \leq n$ , we add index  $i$  to  $I$ . By construction of the set  $I$ , we have  $|I| \leq t$  as required.

We now show that any  $t$  probes of Algorithm `ZeroTestMult` can be perfectly simulated from  $x_{|I}$  and  $b$ . Since the number of probes  $t$  is such that  $t < n$ , we deduce that at least one entire loop (Steps 3 to 5) has not been probed. Let  $j^*$  be the index of this non-probed loop. For all probed variables  $B_i$  between Steps 3 and 5 in loop indices  $j < j^*$ , we have  $i \in I$  and the simulation is straightforward from the input shares  $x_{|I}$ .

It remains to simulate all probed variables between Steps 3 and 5 in loop indices  $j \geq j^*$ , and all probed variables at Step 7. To this aim, we consider two cases whether the output  $b = 0$  or  $b = 1$  (recall that  $b$  is given to the simulator).

If  $b = 1$ , then we know that  $\sum_{i=1}^n B_i = 0 \pmod{q}$  at the end of each for loop. At the end of loop  $j^*$ , since `LinearRefreshMasks` has not been probed, we can perfectly simulate all variables  $B_i$ , by generating random  $B_i$ 's for  $1 \leq i \leq n$  such that  $\sum_{i=1}^n B_i = 0 \pmod{q}$ .

Similarly, if  $b = 0$ , we use the fact that  $u_{j^*}$  has not been probed and acts as a multiplicative one-time pad in  $\mathbb{Z}_q^*$ . This implies that the value encoded by the  $B_i$ 's is randomly distributed in  $\mathbb{Z}_q^*$ . We can therefore perfectly simulate all shares  $B_i$  for  $1 \leq i \leq n$  at the end of loop  $j^*$  by generating random  $B_i$ 's under the condition  $\sum_{i=1}^n B_i \neq 0 \pmod{q}$ .

In both cases, one can propagate the simulation until the end of the for loop, that is until  $j = n$ , and from the knowledge of the  $B_i$  shares at the end of the for loop, one can compute all probed intermediate variables at Step 7 as in the real algorithm. We therefore conclude that the `ZeroTestMult` algorithm is  $(n - 1)$ -Nlo, when the output  $b$  is given to the simulator.

### C.10 Proof of Theorem 2

The `SecExpo` algorithm is  $(n - 1)$ -SNI since it is the composition of several iterations of the `SecMult` algorithm which is  $(n - 1)$ -SNI, with some `RefreshMasks` operations which are also  $(n - 1)$ -SNI. The `ZeroTestExpoShares` algorithm is  $(n - 1)$ -SNI since it is essentially composed by the `SecExpo` algorithm which is  $(n - 1)$ -SNI, where the output variables are simply modified with some known constants. This implies that the `ZeroTestExpo` algorithm is  $(n - 1)$ -Nlo.

## D Polynomial comparison

### D.1 Polynomial comparison of Boolean masked coefficients

We are given as input a set of  $\ell \cdot n$  shares  $(x^{(j)})_i \in \{0, 1\}^k$  for  $1 \leq j \leq \ell$  and  $1 \leq i \leq n$ , corresponding to  $\ell$  coefficients:

$$x^{(j)} = x_1^{(j)} \oplus \dots \oplus x_n^{(j)}$$

and we must output a single bit  $b$  such that  $b = 1$  if  $x^{(j)} = 0$  for all  $1 \leq j \leq \ell$ , and  $b = 0$  otherwise. The simplest approach is to perform a Boolean zero-test of each  $x^{(j)}$  as in Section C.3, keeping each resulting bit  $b^{(j)}$  in Boolean  $n$ -shared form, and then to perform a sequence of `SecAnds` between the bits  $b^{(j)}$ , and to eventually recombine the shares into a bit  $b$ . The complexity of this approach is then  $\mathcal{O}(\ell \cdot n^2 \cdot \log k)$ . A slightly better approach is to high-order compute:

$$y = \bigwedge_{j=1}^{\ell} \overline{x^{(j)}} \in \{0, 1\}^k$$

Then  $y = 0$  iff  $x^{(j)} = 0$  for all  $1 \leq j \leq \ell$ , so we eventually perform a single zero-test of  $y$ . In this approach we take advantage of computing the `SecAnd`s over  $k$  bits instead of a single bit. The complexity is then  $\mathcal{O}(\ell \cdot n^2 + n^2 \cdot \log k)$ . We obtain the pseudo-code below.

---

**Algorithm 21** PolyZeroTestBool

---

**Input:**  $k \in \mathbb{Z}$ , and  $(x_i^{(j)}) \in \{0, 1\}^k$  for  $1 \leq i \leq n$  and  $1 \leq j \leq \ell$ .  
**Output:**  $b \in \{0, 1\}$  with  $b = 1$  if  $\bigoplus_i x_i^{(j)} = 0$  for all  $1 \leq j \leq \ell$ , and  $b = 0$  otherwise

- 1: **for**  $j = 1$  to  $\ell$  **do**  $x_1^{(j)} \leftarrow \overline{x_1^{(j)}}$
- 2:  $(y_1, \dots, y_n) \leftarrow (1, 0, \dots, 0)$
- 3: **for**  $j = 1$  to  $\ell$  **do**  $(y_1, \dots, y_n) \leftarrow \text{SecAnd}(k, (y_1, \dots, y_n), (x_1, \dots, x_n))$
- 4:  $y_1 \leftarrow \overline{y_1}$
- 5: **Return** ZeroTestBoolLog( $k, y_1, \dots, y_n$ )

---

The number of operations is:

$$T_{\text{PolyZeroTestBool}}(k, \ell, n) = \ell \cdot (1 + T_{\text{SecAnd}}(n)) + 1 + T_{\text{ZeroTestBoolLog}}(k, n)$$

The following theorem shows that the adversary does not learn more than the output bit  $b$  of the comparison. The proof is straightforward and therefore omitted.

**Theorem 13.** *The PolyZeroTestBool algorithm is  $(n - 1)$ -NIo.*

## D.2 Polynomial comparison modulo $2^k$ via arithmetic to Boolean conversion

We are given as input a set of  $\ell \cdot n$  shares  $(x^{(j)})_i$  for  $1 \leq j \leq \ell$  and  $1 \leq i \leq n$ , corresponding to  $\ell$  coefficients:

$$x^{(j)} = x_1^{(j)} + \dots + x_n^{(j)} \pmod{q}$$

and we must output a single bit  $b$  such that  $b = 1$  if  $x^{(j)} = 0$  for all  $1 \leq j \leq \ell$ , and  $b = 0$  otherwise. For this we simply perform an arithmetic to Boolean conversion of each coefficient  $x^{(j)}$  separately and then apply the previous PolyZeroTestBool algorithm. The complexity of each Boolean to arithmetic conversion is  $\mathcal{O}(n^2 \cdot k)$  for  $k = \lceil \log_2 q \rceil$ . Therefore the total complexity is  $\mathcal{O}(\ell \cdot n^2 \cdot k)$ . We provide the pseudocode of the corresponding algorithm PolyZeroTestAB below.

---

**Algorithm 22** PolyZeroTestAB

---

**Input:**  $q \in \mathbb{Z}$ ,  $k \in \mathbb{Z}$  with  $q \leq 2^k$ , and  $(x_i^{(j)}) \in \mathbb{Z}_q$  for  $1 \leq i \leq n$  and  $1 \leq j \leq \ell$ .  
**Output:**  $b \in \{0, 1\}$  with  $b = 1$  if  $\sum_i x_i^{(j)} = 0 \pmod{q}$  for all  $1 \leq j \leq \ell$ , and  $b = 0$  otherwise

- 1: **for**  $j = 1$  to  $\ell$  **do**
- 2:  $(y_i^{(j)})_{1 \leq i \leq n} \leftarrow \text{ArithmeticToBoolean}(q, (x_i^{(j)})_{1 \leq i \leq n})$
- 3: **end for**
- 4: **return** PolyZeroTestBool( $k, (y_i^{(j)})$ )

---

The number of operations is

$$T_{\text{PolyZeroTestAB}}(k, n) = \ell \cdot T_{AB}(k, n) + T_{\text{PolyZeroTestBool}}(k, n)$$

The following theorem shows that the adversary does not learn more than the output bit  $b$  of the comparison. The proof is straightforward and therefore omitted.

**Theorem 14.** *The PolyZeroTestAB algorithm is  $(n - 1)$ -NIo.*

### D.3 Polynomial comparison modulo prime $q$ : reduction step

When working modulo a prime  $q$ , we can apply the technique from [BDH<sup>+</sup>21] that efficiently reduces the zero-testing of  $\ell$  coefficients to the zero-testing of  $\kappa \ll \ell$  coefficients, with  $\kappa = \lceil \lambda / \log_2 q \rceil$ , where  $\lambda$  is the security parameter. Given as input  $\ell$  coefficients  $x^{(j)} \in \mathbb{Z}_q$  with arithmetic shares  $x_i^{(j)}$ , the technique consists in computing  $\kappa$  linear combinations:

$$y^{(k)} = \sum_{j=1}^{\ell} a_{kj} \cdot x^{(j)} \bmod q \quad (9)$$

for  $1 \leq k \leq \kappa$ , with randomly distributed coefficients  $a_{kj} \in \mathbb{Z}_q$ . The above equation is actually high-order computed using the arithmetic shares  $x_i^{(j)}$  of each  $x^{(j)}$ , and we obtain the arithmetic shares  $y_i^{(k)}$  of each coefficient  $y^{(k)}$ . We obtain the pseudo-code below.

---

#### Algorithm 23 PolyZeroTestRed [BDH<sup>+</sup>21]

---

**Input:**  $q \in \mathbb{Z}$ , a parameter  $\kappa$  and  $(x_i^{(j)}) \in \mathbb{Z}_q$  for  $1 \leq i \leq n$  and  $1 \leq j \leq \ell$ .

**Output:**  $(y_i^{(k)}) \in \mathbb{Z}_q$  for  $1 \leq i \leq n$  and  $1 \leq k \leq \kappa$ .

```

1: for  $k = 1$  to  $\kappa$  do
2:   for  $i = 1$  to  $n$  do  $y_i^{(k)} \leftarrow 0$ 
3:   for  $j = 1$  to  $\ell$  do
4:      $a_{kj} \leftarrow \mathbb{Z}_q$ 
5:     for  $i = 1$  to  $n$  do  $y_i^{(k)} \leftarrow y_i^{(k)} + a_{kj} \cdot x_i^{(j)}$ 
6:   end for
7: end for
8: return  $(y_i^{(k)})_{1 \leq k \leq \kappa, 1 \leq i \leq n}$ 

```

---

Now if  $x^{(j)} = 0$  for all  $1 \leq j \leq \ell$ , then  $y^{(k)} = 0$  for all  $1 \leq k \leq \kappa$ . If  $x^{(j)} \neq 0$  for some  $1 \leq j \leq \ell$ , then for each  $1 \leq k \leq \kappa$ , we have  $y^{(k)} \neq 0$ , except with probability  $1/q$ . Therefore we must have  $y^{(k)} \neq 0$  for some  $1 \leq k \leq \kappa$ , except with error probability  $q^{-\kappa}$ . We have therefore reduced the zero-testing of  $\ell$  coefficients to the zero-testing of  $\kappa \ll \ell$  coefficients. To reach an error probability  $\leq 2^{-\lambda}$  for security parameter  $\lambda$ , one must take  $\kappa = \lceil \lambda / \log_2 q \rceil$ .

We stress that after this reduction step we cannot zero-test the coefficients  $y^{(k)}$  separately. Otherwise, since the coefficients  $a_{kj}$  in (9) are computed in the clear, knowing that  $y^{(k)} = 0$  for some  $k$  would leak an equation over the coefficients  $x^{(j)}$ , which would leak information about the  $x^{(j)}$  with fewer than  $n$  probes. Instead, the remaining  $\kappa$  coefficients  $y^{(k)}$  must be zero-tested all at once.

This reduction technique is quite efficient because the random coefficients  $a_{kj}$  in (9) are non-masked, which implies that each multiplication  $a_{kj} \cdot x^{(j)}$  can be computed in time  $\mathcal{O}(n)$  for



$n$  shares, instead of  $\mathcal{O}(n^2)$  for a fully masked multiplication. The total complexity of this first step is therefore  $\mathcal{O}(\ell \cdot \kappa \cdot n)$ , with a number of operations:

$$T_{\text{PolyZeroTestRed}}(\kappa, \ell, n) = \kappa \cdot \ell \cdot (2n + 1)$$

**Theorem 15** ([BDH<sup>+</sup>21]). *The PolyZeroTestRed algorithm is  $(n - 1)$ -NI.*

#### D.4 Polynomial comparison modulo $q$ via multiplicative masking

---

##### Algorithm 24 PolyZeroTestMult

---

**Input:**  $q \in \mathbb{Z}$ , a parameter  $\kappa$  and  $(x_i^{(j)}) \in \mathbb{Z}_q$  for  $1 \leq i \leq n$  and  $1 \leq j \leq \ell$ .  
**Output:**  $b \in \{0, 1\}$  with  $b = 1$  if  $\sum_i x_i^{(j)} = 0 \pmod{q}$  for all  $1 \leq j \leq \ell$  and  $b = 0$  otherwise

- 1:  $(y_i^{(k)})_{1 \leq k \leq \kappa, 1 \leq i \leq n} \leftarrow \text{PolyZeroTestRed}((x_i^{(j)})_{1 \leq j \leq \ell, 1 \leq i \leq n})$
- 2:  $b \leftarrow 0$
- 3: **for**  $k = 1$  to  $\kappa$  **do**
- 4:     **for**  $i = 1$  to  $n$  **do**  $z_i \leftarrow 0$
- 5:     **for**  $j = 1$  to  $\ell$  **do**
- 6:         **for**  $i = 1$  to  $n$  **do**  $a_i^{(j)} \leftarrow \mathbb{Z}_q$
- 7:          $(z_1^{(j)}, \dots, z_n^{(j)}) \leftarrow \text{SecMult}((a_1^{(j)}, \dots, a_n^{(j)}), (y_1^{(j)}, \dots, y_n^{(j)}))$
- 8:         **for**  $i = 1$  to  $n$  **do**  $z_i \leftarrow z_i + z_i^{(j)} \pmod{q}$
- 9:     **end for**
- 10:      $b_k \leftarrow \text{ZeroTestMult}(z_1, \dots, z_n)$
- 11:      $b \leftarrow b + b_k$
- 12: **end for**
- 13: **if**  $b = \kappa$  **then**  $b \leftarrow 1$  **else**  $b \leftarrow 0$
- 14: **return**  $b$

---

The complexity of the PolyZeroTestMult algorithm is

$$T_{\text{PolyZeroTestMult}}(q, \kappa, \ell, n) = T_{\text{PolyZeroTestRed}}(\kappa, \ell, n) + \kappa \cdot (\kappa \cdot (T_{\text{SecMult}}(n) + 2n) + T_{\text{ZeroTestMult}}(n))$$

#### D.5 Proof of Theorem 3 (soundness of PolyZeroTestMult)

We denote by `PolyZeroTestMultLoop`, one loop iteration on  $k$  of the `PolyZeroTestMult` algorithm, namely, going from line 4 to 11. We start by showing that `PolyZeroTestMultLoop` computes the correct answer  $b_k$ , except with probability at most  $1/q$ . Indeed, in `PolyZeroTestMultLoop`, one securely computes the value  $z = \sum_{j=1}^{\ell} a^{(j)} \cdot y^{(j)} \pmod{q}$  where the random values  $a^{(j)}$  are uniformly distributed in  $\mathbb{Z}_q$ . Thus, if  $z \neq 0$ , then at least one coefficient  $y^{(j)}$  is not zero and the output  $b_k = 0$  is always correct.

However, if  $z = 0$ , two cases arise: either all coefficients  $y^{(j)}$  are null in which case the algorithm outputs  $b_k = 1$  which is correct, or at least one coefficient  $y^{(j)}$  is such that  $y^{(j)} \neq 0$

but with  $\sum_{j=1}^{\ell} a^{(j)} \cdot y^{(j)} = 0 \pmod{q}$  and the output  $b_k = 1$  in this case is incorrect. Since the  $a^{(j)}$  values are uniformly distributed in  $\mathbb{Z}_q$ , the result of the linear combination of the  $y^{(j)} \neq 0$  with the values  $a^{(j)}$  is also uniform in  $\mathbb{Z}_q$ . Therefore the probability that  $\sum_{j=1}^{\ell} a^{(j)} \cdot y^{(j)} = 0 \pmod{q}$  is  $1/q$  for each iteration of `PolyZeroTestMultLoop`.

Hence by iterating `PolyZeroTestMultLoop`  $\kappa$  times with fresh random values  $a_{\kappa}^{(j)}$  as done in `PolyZeroTestMult`, the probability that  $\sum_{j=1}^{\kappa} a_{\kappa}^{(j)} \cdot y^{(j)} \pmod{q} = 0$  for all  $\kappa$  iterations, with at least one coefficient  $y^{(j)} \neq 0$ , is  $(1/q)^{\kappa} = q^{-\kappa}$ .

## D.6 Proof of Theorem 4 (security of PolyZeroTestMult)

As before, we denote by `PolyZeroTestMultLoop`, one loop iteration on  $k$  of the `PolyZeroTestMult` algorithm (line 4 to 11). We write  $y^{(j)} = \sum_{i=1}^n y_i^{(j)} \pmod{q}$ . We distinguish two cases: either  $y^{(j)} = 0$  for all  $1 \leq j \leq \ell$ , or  $y^{(j)} \neq 0$  for some  $j$ . We show that the simulator can perform a perfect simulation in both cases. Moreover, by assumption the simulator eventually receives the bit  $b$ . This means that the simulator can distinguish the two cases, except with error probability at most  $q^{-\kappa}$ . Therefore the error probability of the simulator will be at most  $q^{-\kappa}$ .

**$y^{(j)} = 0$  for all  $1 \leq j \leq \ell$ .** This is the easy case. Namely in that case, we know that  $b_k = 1$  for all  $k$ . The computation of the shares  $z_i$  at Line 8 is  $(n-1)$ -SNI. Knowing  $b_k$ , the algorithm `ZeroTestMult` at Step 10 is  $(n-1)$ -NI from Theorem 1. Therefore the global `PolyZeroTestMultLoop` algorithm remains  $(n-1)$ -NI.

**$y^{(j)} \neq 0$  for some  $1 \leq j \leq \ell$ .** We consider a sequence of games.

**Game<sub>0</sub>:** we generate all variables as in the algorithm. We assume that we know all input shares  $y_i^{(j)}$ . We can therefore perform a perfect simulation of all probes. Moreover, we have that  $\Pr[b_k = 1] = 1/q$  for all  $1 \leq k \leq \kappa$ , and the variables  $b_k$  are independently distributed.

**Game<sub>1</sub>:** we modify the way the variables are generated. Instead of generating all variables  $a_i^{(j)}$  uniformly and independently, we first generate the bits  $b_k$  independently with  $\Pr[b_k = 1] = 1/q$ . Then for each  $1 \leq k \leq \kappa$ , if  $b_k = 1$  then we generate the shares  $a_i^{(j)}$  such that  $\sum_{j=1}^{\ell} a^{(j)} y^{(j)} = 0 \pmod{q}$ , where  $a^{(j)} = \sum_{i=1}^n a_i^{(j)} \pmod{q}$ . Otherwise, we generate the shares  $a_i^{(j)}$  such that  $\sum_{j=1}^{\ell} a^{(j)} y^{(j)} \neq 0 \pmod{q}$ . The distribution of the variables is the same as in the previous game. Therefore, we can still perform a perfect simulation of all probed variables.

**Game<sub>2</sub>:** we show that we can still perform a perfect simulation as in **Game<sub>1</sub>**, but only with the input shares  $y_{|I}^{(j)}$  for a subset  $|I| \leq t$ . This will prove that the algorithm is  $(n-1)$ -NI.

Firstly, from the  $(n-1)$ -SNI property of `SecMult` and the  $(n-1)$ -NI property of `ZeroTestMult` knowing  $b_k$ , the simulation of all probes can be performed from the knowledge of a subset  $y_{|I}^{(j)}$  of the input shares for  $|I| \leq t$ , and a subset  $a_{|J}^{(j)}$  of the shares of the values  $a^{(j)}$ , for  $|J| \leq t \leq n-1$ . Secondly, the constraints on  $\sum_{j=1}^{\ell} a^{(j)} y^{(j)}$  from **Game<sub>2</sub>** can be satisfied by generating all shares  $a_i^{(j)}$  for  $i \neq i^*$  uniformly at random, and by fixing  $a_{i^*}^{(j)}$ , without changing the distribution of the shares  $a_i^{(j)}$ , for some  $i^* \notin J$ . Finally, since the knowledge of  $a_{i^*}^{(j)}$  is not needed for the simulation, we can perform a perfect simulation of all probes from  $y_{|I}^{(j)}$ . This concludes the proof.

Note that a  $n$ -sharing of the coefficients  $a^{(j)}$  is required for the simulation. If the coefficients  $a^{(j)}$  were computed in the clear, one could not satisfy the constraints on the linear sums, without knowing the coefficients  $y^{(j)}$ .

## D.7 Polynomial comparison modulo $q$ via exponentiation

---

### Algorithm 25 PolyZeroTestExpo

---

**Input:**  $q \in \mathbb{Z}$ ,  $\kappa \in \mathbb{Z}$ , and  $(x_i^{(j)}) \in \mathbb{Z}_q$  for  $1 \leq i \leq n$  and  $1 \leq j \leq \ell$ .

**Output:**  $b \in \{0, 1\}$  with  $b = 1$  if  $\sum_i x_i^{(j)} = 0 \pmod{q}$  for all  $1 \leq j \leq \ell$  and  $b = 0$  otherwise

```

1:  $(y_i^{(k)})_{1 \leq k \leq \kappa, 1 \leq i \leq n} \leftarrow \text{PolyZeroTestRed}((x_i^{(j)})_{1 \leq j \leq \ell, 1 \leq i \leq n})$ 
2:  $(B_1, \dots, B_n) \leftarrow \text{ZeroTestExpoShares}(y_1^{(1)}, \dots, y_n^{(1)})$ 
3: for  $j = 2$  to  $\kappa$  do
4:    $(C_1, \dots, C_n) \leftarrow \text{ZeroTestExpoShares}(y_1^{(j)}, \dots, y_n^{(j)})$ 
5:    $(B_1, \dots, B_n) \leftarrow \text{SecMult}((B_1, \dots, B_n), (C_1, \dots, C_n))$ 
6: end for
7:  $(C_1, \dots, C_n) \leftarrow \text{RefreshMasks}(B_1, \dots, B_n)$ 
8:  $B \leftarrow C_1 + \dots + C_n \pmod{q}$ 
9: if  $B = 1$  then return 1
10: else return 0

```

---

The number of operations is:

$$T_{\text{PolyZeroTestExpo}}(q, \kappa, \ell, n) = T_{\text{PolyZeroTestRed}}(\kappa, \ell, n) + \kappa \cdot T_{\text{ZeroTestExpoShares}}(q, n) + (\kappa - 1) \cdot T_{\text{SecMult}}(n) + T_{\text{RefreshMasks}}(n) + n$$

## E Polynomial comparison for Kyber

### E.1 Proof of Lemma 1

We first show that if  $x \in \text{Compress}_{q,d}^{-1}(\tilde{c})$ , then we must have  $|x - y \pmod{\pm q}| \leq B_{q,d}$ , where  $y = \text{Decompress}_{q,d}(\tilde{c})$ . This will imply:

$$\text{Compress}_{q,d}^{-1}(\tilde{c}) \subset [y - B_q, \dots, y + B_q]_q$$

Namely in this case we have  $\tilde{c} = \text{Compress}_{q,d}(x)$ , and therefore we have

$$y = \text{Decompress}_{q,d}(\text{Compress}_{q,d}(x)) = \left\lfloor \frac{q}{2^d} \cdot \left( \left\lceil \frac{2^d}{q} \cdot x \right\rceil \pmod{2^d} \right) \right\rfloor$$

We write  $\left\lceil (2^d/q) \cdot x \right\rceil = (2^d/q) \cdot x + \varepsilon$  for some  $|\varepsilon| \leq 1/2$ . We obtain:

$$\begin{aligned} |x - y \pmod{\pm q}| &= \left| x - \left\lfloor \frac{q}{2^d} \cdot \left( \left\lceil \frac{2^d}{q} \cdot x \right\rceil \pmod{2^d} \right) \right\rfloor \pmod{\pm q} \right| \\ &= \left| x - \left\lfloor \frac{q}{2^d} \cdot \left( \frac{2^d}{q} \cdot x + \varepsilon \pmod{2^d} \right) \right\rfloor \pmod{\pm q} \right| \\ &= \left| x - \left[ x + \frac{q}{2^d} \cdot \varepsilon \pmod{q} \right] \pmod{\pm q} \right| = \left| \left\lfloor \frac{q}{2^d} \cdot \varepsilon \right\rfloor \right| \leq B_{q,d} \end{aligned}$$

Conversely, we show that if  $|x - y \bmod^\pm q| \leq B_{q,d} - 1$ , then we must have  $x \in \text{Compress}_{q,d}^{-1}(\tilde{c})$ . This will imply  $[y - B_{q,d} + 1, \dots, y + B_{q,d} - 1]_q \subset \text{Compress}_{q,d}^{-1}(\tilde{c})$ . Namely we write again  $\lfloor (2^d/q) \cdot x \rfloor = (2^d/q) \cdot x + \varepsilon$  for some  $|\varepsilon| \leq 1/2$ , and we can write:

$$\begin{aligned} \left| \tilde{c} - \text{Compress}_{q,d}(x) \bmod^\pm 2^d \right| &= \left| \tilde{c} - \left( \left\lfloor \frac{2^d}{q} \cdot x \right\rfloor \bmod 2^d \right) \bmod^\pm 2^d \right| \\ &= \left| \tilde{c} - \left( \frac{2^d}{q} \cdot x + \varepsilon \right) \bmod^\pm 2^d \right| \\ &= \frac{2^d}{q} \cdot \left| \frac{q}{2^d} \cdot \tilde{c} - x - \frac{q}{2^d} \cdot \varepsilon \bmod^\pm q \right| \end{aligned}$$

We write  $y = \lfloor (q/2^d) \cdot \tilde{c} \rfloor = (q/2^d) \cdot \tilde{c} + \varepsilon'$  for some  $|\varepsilon'| \leq 1/2$ , which gives:

$$\begin{aligned} \left| \tilde{c} - \text{Compress}_{q,d}(x) \bmod^\pm 2^d \right| &= \frac{2^d}{q} \cdot \left| y - \varepsilon' - x - \frac{q}{2^d} \cdot \varepsilon \bmod^\pm q \right| \\ &\leq \frac{2^d}{q} \cdot \left( B_{q,d} - 1 + \frac{q}{2^{d+1}} + \frac{1}{2} \right) < 1 \end{aligned}$$

For the last inequality we use  $B_{q,d} < q/(2^{d+1}) + 1/2$  for odd  $q$ . This implies  $\tilde{c} = \text{Compress}_{q,d}(x)$ , which proves the lemma.

**Generating the list of candidates.** From Lemma 1, to generate the list of candidates  $\text{Compress}_{q,d}^{-1}(\tilde{c})$ , it suffices to consider the set  $[a, b]_q$  with  $a = y - B_{q,d}$  and  $b = y + B_{q,d}$  and to test whether the two elements at the border belong to the set, that is we check whether  $\text{Compress}_{q,d}(a) = \tilde{c}$  and  $\text{Compress}_{q,d}(b) = \tilde{c}$ . We provide the pseudocode in Algorithm 26 below.

---

**Algorithm 26** CompressInv

---

**Input:**  $\tilde{c} \in \mathbb{Z}_{2^k}$

**Output:**  $a, b \in \mathbb{Z}$  such that  $\text{Compress}_{q,d}^{-1}(\tilde{c}) = [a, b]_q$ .

- 1:  $B_{q,d} \leftarrow \lfloor \frac{q}{2^{d+1}} \rfloor$
  - 2:  $y \leftarrow \text{Decompress}_{q,d}(\tilde{c})$
  - 3:  $a \leftarrow y - B_{q,d} \bmod q, b \leftarrow y + B_{q,d} \bmod q$
  - 4: **if**  $\text{Compress}_{q,d}(a) \neq \tilde{c}$  **then**  $a \leftarrow a + 1 \bmod q$
  - 5: **if**  $\text{Compress}_{q,d}(b) \neq \tilde{c}$  **then**  $b \leftarrow b - 1 \bmod q$
  - 6: **return**  $a, b$
-

## E.2 The SecMultList algorithm

---

### Algorithm 27 SecMultList

---

**Input:**  $x_1, \dots, x_n \in \mathbb{Z}_q$  s.t.  $\sum_i x_i \pmod{q} = x$ , a prime  $q$ , an index  $m$  and  $a_1, \dots, a_m \in \mathbb{Z}_q$

**Output:**  $z_1, \dots, z_n \in \mathbb{Z}_q$  such that  $\sum_{i=1}^n z_i = \prod_{i=1}^m (x - a_i) \pmod{q}$

1:  $(z_1, z_2, \dots, z_n) \leftarrow (1, 0, \dots, 0)$

2: **for**  $i = 1$  to  $m$  **do**

3:      $(z_1, z_2, \dots, z_n) \leftarrow \text{SecMult}((z_1, \dots, z_n), (x_1 - a_i \pmod{q}, x_2, \dots, x_n))$

4: **end for**

5: **return**  $(z_1, \dots, z_n)$

---

For  $m$  candidates, the number of operations for high-order computing  $z$  is therefore at most:

$$T_{\text{SecMultList}}(d, n) = (2 \cdot B_{q,d} + 1) \cdot T_{\text{SecMult}}(n)$$

**Theorem 16.** *The SecMultList algorithm is  $(n - 1)$ -SNI.*

*Proof.* The SecMultList algorithm is  $(n - 1)$ -SNI since it is the composition of  $m$  iterations of the  $(n - 1)$ -SNI secMult algorithm. We stress that the first multiplication by 1 (initialized in line 1) is here on purpose since it is equivalent to an  $(n - 1)$ -SNI masks refreshing, which ensures the independence between both subsequent inputs.  $\square$

## E.3 Polynomial comparison with range test

We recall the technique from [BGR<sup>+</sup>21] for testing whether  $x \in [a, b]_q$ , where  $[a, b]_q = \text{Compress}_{q,d}^{-1}(\tilde{c})$ , by performing two high-order comparisons with the interval bounds  $a$  and  $b$ .

We let  $k = \lfloor \log_2 q \rfloor$ , so that  $2^k < q < 2^{k+1}$ . We write  $\Delta := q - 2^k - 1$ . For Kyber with  $q = 3329$ , we get  $k = 11$  and  $\Delta = 1280$ . We assume that the bounds of the interval  $[a, b]_q$  satisfy  $b - a \pmod{\pm q} \leq \Delta$ . Recall that we have the upper-bound  $b - a \pmod{\pm q} \leq 2 \cdot B_{q,d} + 1 = 2 \cdot \lfloor q/2^{d+1} \rfloor + 1$ . Therefore the assumption is satisfied for Kyber for  $d \geq 2$ .

Taking as input the shares  $x_i$  of  $x = x_1 + \dots + x_n \pmod{q}$ , we want to output a  $n$ -shared bit  $u$  such that  $u = 1$  if  $x \in [a, b]_q$  and  $u = 0$  otherwise. For this we use:

$$x \in [a, b]_q \iff ((2^k + x - a \pmod{q}) \geq 2^k) \wedge ((x - b - 1 \pmod{q}) \geq 2^k) \quad (10)$$

Namely we have using  $\Delta = q - 2^k - 1$ :

$$((2^k + x - a \pmod{q}) \geq 2^k) \iff x \in [a, a + \Delta]_q$$

$$((x - b - 1 \pmod{q}) \geq 2^k) \iff x \in [b - \Delta, b]_q$$

Since by assumption  $b - a \pmod{\pm q} \leq \Delta$ , we have  $[a, b]_q \subset [a, a + \Delta]_q$  and similarly  $[a, b]_q \subset [b - \Delta, b]_q$ . Moreover from  $2\Delta < q$  we must have  $[b, a + \Delta]_q \cap [b - \Delta, a]_q = \emptyset$ . This implies  $[a, b]_q = [a, a + \Delta]_q \cap [b - \Delta, b]_q$ , which proves (10).

From (10), we perform a high-order arithmetic modulo  $q$  to Boolean conversion of the two values  $2^k + x - a$  and  $x - b - 1$  modulo  $q$  using [BBE<sup>+</sup>18], and we perform a high-order And (with

SecAnd) of the most significant bit of the two results (using the Boolean shares). The number of operations is therefore:

$$T_{\text{range}}(k, n) = 2 \cdot T_{\text{AB}}(k + 1, n) + T_{\text{SecAnd}}(n)$$

---

**Algorithm 28** RangeTestShares

---

**Input:**  $x_1, \dots, x_n \in \mathbb{Z}_q$  for prime  $q$  with  $\sum_i x_i = x \pmod{q}$ ,  $k = \lfloor \log_2 q \rfloor$ , bounds  $a$  and  $b$  s.t.  $b - a \pmod{q} \leq q - 2^k - 1$ .

**Output:**  $u_1, \dots, u_n \in \mathbb{Z}_q$  with  $\sum_i u_i = 1 \pmod{q}$  if  $x \in [a, b]_q$  and  $\sum_i u_i = 0 \pmod{q}$  otherwise

1:  $(A_1, \dots, A_n) \leftarrow (x_1 + 2^k - a \pmod{q}, x_2, \dots, x_n)$

2:  $(B_1, \dots, B_n) \leftarrow (x_1 - b - 1 \pmod{q}, x_2, \dots, x_n)$

3:  $(y_1, \dots, y_n) \leftarrow \text{ArithmeticToBoolean}(q, (A_1, \dots, A_n))$

4:  $(z_1, \dots, z_n) \leftarrow \text{ArithmeticToBoolean}(q, (B_1, \dots, B_n))$

5:  $(u_1, \dots, u_n) \leftarrow \text{SecAnd}(1, (\text{MSB}(y_1), \dots, \text{MSB}(y_n)), (\text{MSB}(z_1), \dots, \text{MSB}(z_n)))$

6: **return**  $(u_1, \dots, u_n)$

---

#### E.4 Number of operations for hybrid approach in polynomial comparison

The number of operations of the polynomial comparison in Kyber is then:

$$T = \ell_1 \cdot T_{\text{SecMultList}}(d_u, n) + \ell_2 \cdot T_{\text{HOComp}}(d_v, n) + \ell_2 + T_{\text{PolyZeroTestBool}}(13, \ell_2, n) + T_{\text{BA}}(1, n) + T_{\text{polyZT}}(q, \ell_1 + 1, n)$$

## F Ring-LWE IND-CPA encryption

We provide an overview of ring-LWE encryption [LPR10]. For  $N \in \mathbb{Z}$  and  $q \in \mathbb{Z}$ , let  $\mathcal{R}$  and  $\mathcal{R}_q$  denote the rings  $\mathbb{Z}[X]/(X^N + 1)$  and  $\mathbb{Z}_q[X]/(X^N + 1)$  respectively. Let  $a \in \mathcal{R}_q$  be a public random polynomial. Let  $\chi$  be a distribution outputting “small” elements in  $\mathcal{R}$ , and let  $s, e \leftarrow \chi$ . The public-key is  $t = as + e \in \mathcal{R}_q$ , while the secret key is  $s$ . To CPA-encrypt a message  $m \in \mathcal{R}$  with binary coefficients, one computes the ciphertext  $(c_1, c_2)$  where

$$\begin{aligned} c_1 &= a \cdot e_1 + e_2 \\ c_2 &= t \cdot e_1 + e_3 + \lfloor q/2 \rfloor \cdot m \end{aligned} \tag{11}$$

with  $e_1, e_2, e_3 \leftarrow \chi$ . To decrypt a ciphertext  $(c_1, c_2)$ , one first computes  $u = c_2 - s \cdot c_1$ , which gives:

$$\begin{aligned} u &= (a \cdot s + e) \cdot e_1 + e_3 + \lfloor q/2 \rfloor \cdot m - s \cdot a \cdot e_1 - s \cdot e_2 \\ &= \lfloor q/2 \rfloor \cdot m + e \cdot e_1 + e_3 - s \cdot e_2 \end{aligned}$$

Since the ring elements  $e, e_1, e_2, e_3$  and  $s$  are small, and the message  $m \in \mathcal{R}$  has binary coefficients, we can recover  $m$  by rounding. Namely, for each coefficient of the above polynomial  $u$ , we decode to 0 if the coefficient is closer to 0 than  $\lfloor q/2 \rfloor$ , and to 1 otherwise. More precisely, we decode the message  $m$  as  $m = \text{th}(c_2 - s \cdot c_1)$ , where  $\text{th}$  applies coefficient-wise the threshold function  $\text{th} : \mathbb{Z}_q \rightarrow \{0, 1\}$ :

$$\text{th}(x) = \begin{cases} 0 & \text{if } x \in (0, q/4) \cup (3q/4, q) \\ 1 & \text{if } x \in (q/4, 3q/4) \end{cases} \tag{12}$$

## G Fully masked implementation of Saber

### G.1 The Saber Key Encapsulation Mechanism (KEM)

Saber [BMD<sup>+</sup>21] is based on the hardness on the module learning-with-rounding (M-LWR) problem. The difference with Kyber is that instead of explicitly adding error terms  $\mathbf{e}, \mathbf{e}_1, e_2$  from a “small” distribution, the errors are deterministically added by applying a rounding function mapping  $\mathbb{Z}_q$  to  $\mathbb{Z}_p$  with  $p < q$ . For Saber, both  $p$  and  $q$  are powers of two; therefore the rounding function is a shift extracting the  $\log_2(p)$  most significant bits of its input.

The Saber submission provides three parameters sets LightSaber, Saber and FireSaber with claimed security level equivalent to AES-128, AES-192 and AES-256 respectively; see Table 14. We recall the pseudocode below. The constants  $h_1, h_2$  and  $\mathbf{h}$  are needed to center the errors introduced by rounding around 0. We write  $q = 2^{\epsilon_q}$  and  $p = 2^{\epsilon_p}$ .

---

#### Algorithm 29 Saber.CPA.KeyGen()

---

- 1:  $\rho, \sigma \leftarrow \{0, 1\}^{256}$
  - 2:  $\mathbf{A} \leftarrow \mathcal{R}_q^{k \times k} := \text{Sam}(\rho)$
  - 3:  $\mathbf{s} \leftarrow \chi_\mu^k := \text{Sam}(\sigma)$
  - 4:  $\mathbf{t} := (\mathbf{A}^T \mathbf{s} + \mathbf{h} \bmod q) \gg (\epsilon_q - \epsilon_p) \in \mathcal{R}_p^k$
  - 5: **return**  $pk := (\mathbf{t}, \rho), sk := \mathbf{s}$
- 

	$N$	$k$	$q$	$p$	$T$	$\mu$
LightSaber	256	2	$2^{13}$	$2^{10}$	$2^3$	5
Saber	256	3	$2^{13}$	$2^{10}$	$2^4$	4
FireSaber	256	4	$2^{13}$	$2^{10}$	$2^6$	3

Table 14: Parameter set for Saber.

---

#### Algorithm 30 Saber.CPA.Enc( $pk, m$ )

---

- 1:  $r \leftarrow \{0, 1\}^{256}$
  - 2:  $\mathbf{A} \leftarrow \mathcal{R}_q^{k \times k} := \text{Sam}(\rho)$
  - 3:  $\mathbf{r} \leftarrow \chi_\mu^k := \text{Sam}(r)$
  - 4:  $\mathbf{u} := (\mathbf{A} \mathbf{r} + \mathbf{h} \bmod q) \gg (\epsilon_q - \epsilon_p) \in \mathcal{R}_p^k$
  - 5:  $v' := \mathbf{t}^T (\mathbf{r} \bmod p) \in \mathcal{R}_p$
  - 6:  $c_m := (v' + h_1 - 2^{\epsilon_p - 1} m \bmod p) \gg (\epsilon_p - \epsilon_T) \in \mathcal{R}_T$
  - 7: **return**  $c := (\mathbf{u}, c_m)$
- 

---

#### Algorithm 31 Saber.CPA.Dec( $sk, c = (\mathbf{u}, c_m)$ )

---

- 1:  $v := \mathbf{u}^T (\mathbf{s} \bmod p) \in \mathcal{R}_p$
  - 2:  $m := (v - 2^{\epsilon_p - \epsilon_T} c_m + h_2 \bmod p) \gg (\epsilon_p - 1) \in \mathcal{R}_2$
  - 3: **return**  $m$
- 

The Saber CCA-secure key encapsulation mechanism is similar to that of Kyber. We recall the pseudocode below.

---

#### Algorithm 32 Saber.Encaps(pk)

---

- 1:  $m \leftarrow \{0, 1\}^{256}$
  - 2:  $(\hat{K}, r) := G(H(pk), m)$
  - 3:  $c := \text{Saber.CPA.Enc}(pk, m; r)$
  - 4:  $K = H(\hat{K}, c)$
  - 5: **return**  $c, K$
- 

---

#### Alg. 33 Saber.Decaps( $sk = (\mathbf{s}, z, \mathbf{t}, \rho), c$ )

---

- 1:  $m' := \text{Saber.CPA.Dec}(\mathbf{s}, c)$
  - 2:  $(\hat{K}', r') := G(H(pk), m')$
  - 3:  $c' := \text{Saber.CPA.Enc}((\mathbf{t}, \rho), m'; r')$
  - 4: **if**  $c = c'$  **then return**  $K := H(\hat{K}', c)$
  - 5: **else return**  $K := H(z, c)$
-

## G.2 High-order masking of Saber

The high-order masking of **Saber** is quite similar to that of **Kyber**. The main difference is that we work with power-of-two moduli. We describe the high-order masking of the **Saber.Decaps** algorithm recalled above (Algorithm 33), using the techniques from the previous sections.

1. We consider Line 1 of Algorithm 33. As previously, we assume that the secret key  $\mathbf{s} \in \mathcal{R}_q^k$  is initially arithmetically masked with  $n$  shares. By modular reduction, we obtain  $n$  shares in  $\mathcal{R}_p^k$ . Therefore, at Line 1 of the **Saber.CPA.Dec** algorithm, we obtain a value  $v$  that is arithmetically  $n$ -shared modulo  $p$ . At Line 2, we obtain  $n$  Boolean shares of the message  $m$  by arithmetic to Boolean conversion modulo  $p = 2^{\epsilon_p}$  using [CGV14], and taking the MSB of each share.
2. At Line 2 of Algorithm 33, starting from the Boolean masked  $m'$ , we use an  $n$ -shared Boolean implementation of the hash function  $G$ , and obtain as output the Boolean  $n$ -shared values  $\hat{K}'$  and  $r'$ .
3. At Line 3 of Algorithm 33, we start with Line 3 of Algorithm 30 which is the masked binomial sampling. Starting from the Boolean  $n$ -shared  $r'$ , we must obtain a value  $\mathbf{r}$  which is arithmetically  $n$ -shared modulo  $q$ . For this we use the  $n$ -masked binomial sampling from [CGMZ22], based on Boolean to arithmetic modulo  $q$  conversion (based on table recomputation).
4. We proceed with Line 4 of Algorithm 30. The vector  $\mathbf{Ar} + \mathbf{h} \bmod q$  is  $n$ -shared modulo  $q$ . We convert from arithmetic to Boolean masking using [CGV14], and then perform a right shift of all Boolean shares by  $\epsilon_q - \epsilon_p$ . The vector  $\mathbf{u}'$  as output of Line 4 of Algorithm 30 is therefore Boolean masked with  $n$  shares.
5. At Line 5, the value  $\mathbf{r}$  is arithmetically masked modulo  $p$  by modular reduction modulo  $p$  of the shares modulo  $q$ . The value  $v'$  is therefore arithmetically masked modulo  $p$ . This enables to compute the value  $v' + h_1 - 2^{\epsilon_p - 1}m \bmod p$  at Line 6 with  $n$  shares modulo  $p$ . As previously the shift by  $\epsilon_p - \epsilon_T$  bits is computed via arithmetic to Boolean conversion. At Line 3 of Algorithm 33, the vector  $\mathbf{u}'$  and the value  $c'_m$  of the ciphertext  $c' = (\mathbf{u}', c'_m)$  are therefore both in Boolean masked form.
6. For the ciphertext comparison at Line 4, we use the same technique as in Section 5.3 for the ciphertext comparison of **Kyber**, for the second part of the ciphertext with the **Compress** function (lines 3 and 4). Eventually we recombine the shares and we obtain a bit  $b$  in the clear, with  $b = 1$  if the two ciphertexts match.
7. Finally, as in **Kyber**, if  $b = 1$ , we use the Boolean  $n$ -shared  $\hat{K}'$  to obtain a Boolean  $n$ -shared session key  $K$ , using an  $n$ -shared implementation of  $H$ . Similarly, if  $b = 0$ , we use the Boolean  $n$ -shared secret  $z$  to obtain the Boolean  $n$ -shared session key  $K$ .