High-order Polynomial Comparison and Masking Lattice-based Encryption

Jean-Sébastien Coron[1], François Gérard[1], Simon Montoya[2,3], and Rina Zeitoun[2]

[1] University of Luxembourg
jean-sebastien.coron@uni.lu, francois.gerard@uni.lu
[2] IDEMIA, Cryptography & Security Labs, Courbevoie, France
simon.montoya@idemia.com, rina.zeitoun@idemia.com
[3] 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 finalists Kyber and Saber, with a concrete implementation.

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 third round, which includes the evaluation of the implementation results of the remaining candidates. In this paper, we consider the two lattice-based finalists Kyber [BDK+18,ABD+21] and Saber [BMD+21], whose security is based on the hardness of the module learning-with-errors (M-LWE) and the module learning-with-rounding (M-LWR) problems, respectively. Those problems are 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 as vulnerable to side-channel attacks as any other cryptosystems, see for example [PPM17,HCY20,XPRO20]. Resistance against side-channel attacks is therefore an explicit criteria for standardization by NIST [MAA+20]. The main countermeasure against side-channel attacks is masking. It consists in splitting every variable x into n shares with $x = x_1 + \cdots + x_n$, and processing the shares separately, so that 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+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 for power-of-two moduli [CGV14], and have been recently extended to prime moduli in [BBE+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. 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'.

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+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+21] also considered a similar attack against the high-order polynomial comparison from [BPO+20], which processed successive blocks of ℓ bits, and the pass/fail bit was computed in the clear for each block, which also led to a CCA attack. The authors of [BDH+21] also proposed a correction of the flaw in [BPO+20], based on computing linear combinations over all coefficients of the polynomial (instead of partial subsets of the coefficients); however, they do not provide a complete description of the high-order comparison between two polynomials.

The above attacks show that the polynomial comparison 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.

In [BGR+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 $\mathsf{th}(x)$ outputting 0 if $x < q/2$ and 1 otherwise. They show that $\mathsf{th}(x) = x_{11} \oplus (\neg x_{11} \cdot x_{10} \cdot x_9 \cdot (x_8 \oplus (\neg x_8 \cdot x_7)))$, 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 $\mathsf{th}(x)$ can be performed by first converting the masking of x from arithmetic modulo q to Boolean, using [BBE+18]; then $\mathsf{th}(x)$ 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+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 re-encryption, the ciphertext comparison requires to check that every coefficient belongs to a certain public range modulo q, instead of checking for equality.

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 [CGMZ21], the authors described high-order table-based conversion algorithms between arithmetic and Boolean masking, based on the randomized table countermeasure from [Cor14]. For converting from Boolean to arithmetic masking, as required for 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 described a modulus switching technique combined with a fast table-based arithmetic to Boolean conversion, that offers significant efficiency improvement compared to previous techniques.

Our contributions. In this paper, we focus on the high-order polynomial comparison (Step 3). We consider four techniques, firstly for zero testing a single coefficient, and then zero testing a set of polynomials at once. The first two techniques are part of the state of the art, while the last two are new. We refer to Table 1 for a summary, with the corresponding asymptotic complexities. The first technique is straightforward: it starts from a Boolean masked ciphertext and consists in performing the comparison via a high-order masked Boolean circuit. Similarly, the second technique starts from an arithmetic masking modulo 2^k or a prime q, and performs an arithmetic to Boolean conversion so that the comparison can be performed over Boolean shares as previously. Our third technique is based on masked exponentiation modulo a prime q, using Fermat's little theorem. Finally, our fourth technique is based on converting from arithmetic masking modulo q to multiplicative masking, which enables to perform a zero-test of x without revealing more information about x.

	zero-testing	Multi coef.	Masking	Complexity
PolyZeroTestBool	Secure And	SecAnd	Boolean	$\mathcal{O}(\ell n^2 + n^2 \log k)$
PolyZeroTestAB	A $\to$ B conv.	SecAnd	mod $q, 2^k$	$\mathcal{O}(\ell n^2 \log k)$
PolyZeroTestExpo	Exponentiation	SecMult	mod q	$\mathcal{O}(\ell \kappa n + \kappa n^2 \log q)$
PolyZeroTestMult	Mult. masking	Linear comb.	mod q	$\mathcal{O}(\ell \kappa n + \kappa^2 n^2)$

Table 1. Complexities of polynomial comparison, with ℓ 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 λ is the security parameter.

As an application, we consider the high-order polynomial comparison in Kyber for IND-CCA decryption. Recall that in Kyber the ciphertext coefficients are compressed from modulo q to d bits by computing $\mathsf{Compress}_{q,d}(x) := \lfloor (2^d/q) \cdot x \rceil \bmod 2^d$. We first consider an alternative approach to [BGR+21], where we explicitly high-order mask the Compress function during the encryption process. For this we extend the modulus switching technique from [FBR+21] which was first-order only. To our knowledge, this is the first proposal for high-order masking the Compress function of Kyber[1]. We also consider the high-order ciphertext comparison without the Compress function as in [BGR+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. 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+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 and Saber schemes 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

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 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 $\lfloor x \rceil$ 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+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+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*

[1] In [BGR+21] and [CGMZ21], the masking of the Compress function was considered only for $d = 1$ bit as output, as used in IND-CPA decryption.

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 $x_{|I}$.

However, in our paper, even the SNI definition is not quite sufficient to prove the security of our constructions. Namely, in the context of 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. Clearly that would break the SNI definition, since for $(n-1)$-SNI security, at most $n-1$ output shares b_i can be simulated. Instead, we can assume that the output bit b of the comparison is given for free to the simulator, since that bit b is known by the adversary. In that case, using an extension of the SNI definition, we show that all n output shares b_i can be perfectly simulated, which in turn enables to properly simulate the recombination of the b_i's into b.

More precisely, 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 B.2) satisfies the extended notion.

Definition 3 (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 1 ([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 $b = b_1 + \cdots + b_n$. We can then recombine the output shares of the gadget, and all intermediate variables in the recombination can be perfectly simulated. This shows that the resulting gadget satisfies the NI property.[2]

Lemma 2. *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 $b = b_1 + \cdots + b_n$.*

Proof. We use the set I obtained from Definition 3. If $|I| = n$, we can simulate all output variables. Otherwise, let $i^\star \notin I$. We let O such that $I \cup O = [1,n] \setminus \{i^\star\}$. From the free-$t$-SNI definition, we can simulate all variables in $b_{I\cup O}$. The remaining variable $b_{i^\star}$ is simulated from the knowledge of b as $b_{i^\star} = b - \sum_{i\neq i^\star} b_i$. □

Corollary 1 (NI security). *Let G be a gadget taking as input n shares $(a_i)_{1\leq i\leq n}$ and outputting n shares $(b_i)_{1\leq i\leq n}$, and let G' be the same as G but outputting $b = b_1 + \cdots + b_n$. Assume*

[2] 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.

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.

3 High-order zero testing

In the IND-CCA decryption of lattice-based schemes, 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+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 $x = x_1 \oplus \cdots \oplus x_n \in \{0,1\}^k$, or the n arithmetic shares of $x = x_1 + \cdots + x_n \bmod q$, 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 Section 3.1 (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; we recall the technique in Section 3.2 (ZeroTestAB algorithm). For arithmetic shares modulo a prime q, we describe two new zero-testing algorithms. The first technique (ZeroTestExpo in Section 3.3) 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. The second technique (ZeroTestMult in Section 3.4) 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. We will see in Section 4 that for zero testing ℓ coefficients at once, these two techniques are much more efficient than arithmetic to Boolean conversion. We refer to Table 2 for a summary.

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 ZeroTestExpo or ZeroTestMult, 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 with the (much faster) ZeroTestExpo and ZeroTestMult algorithms.

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

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

3.1 Boolean zero testing in $\{0,1\}^k$

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

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

Starting from the n Boolean shares of $x = x_1 \oplus \cdots \oplus x_n$, the right-hand side of the above equation can be computed by a sequence of $k-1$ secure And; we recall in Appendix B.1 the SecAnd algorithm. 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 $b_1, \ldots, b_n$ are eventually recombined after a mask refreshing; we refer to Appendix B.2 for a description of RefreshMasks. We obtain Algorithm 1 below.

Algorithm 1 ZeroTestBool

Input: $k \in \mathbb{N}$ and $x_1, \ldots, x_n \in \{0,1\}^k$
Output: $b \in \{0,1\}$ such that $b = 1$ if $\oplus x_i = 0$, and $b = 0$ otherwise.
1: $(y_1, \ldots, y_n) \leftarrow (\overline{x_1}, x_2, \ldots, x_n)$
2: $(b_1, \ldots, b_n) \leftarrow (1, 0, \ldots, 0)$
3: **for** $j = 0$ to $k-1$ **do**
4: $\quad (b_1, \ldots, b_n) \leftarrow \mathsf{SecAnd}(1, (b_1, \ldots, b_n), ((y_1 \gg j)\ \&\ 1, \ldots, (y_n \gg j)\ \&\ 1))$
5: **end for**
6: $(b_1, \ldots, b_n) \leftarrow \mathsf{RefreshMasks}(b_1, \ldots, b_n)$
7: **return** $b_1 \oplus \cdots \oplus b_n$

Theorem 1. *The* ZeroTestBool *is* $(n-1)$-NI*, when* b *is given to the simulator.*

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 RefreshMasks, the ZeroTestBool algorithm up to Line 6 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. □

Improved $\mathcal{O}(n^2 \cdot \log k)$ complexity. For a k-bit input x, the above algorithm has complexity $\mathcal{O}(n^2 \cdot k)$. In Appendix B.3, 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.

3.2 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 $x = x_1 + \cdots + x_n \bmod q$, 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 $y_1, \ldots, y_n \in \{0,1\}^k$, with $x = y_1 \oplus \cdots \oplus y_n$. We then apply the Boolean zero-testing algorithm from the previous section. We obtain the pseudo-code below.

Algorithm 2 ZeroTestAB

Input: $q \in \mathbb{Z}$, $k \in \mathbb{Z}$ with $q \leq 2^k$, and $x_1, \ldots, x_n \in \mathbb{Z}_q$
Output: $b \in \{0,1\}$ with $b = 1$ if $\sum_i x_i = 0 \pmod q$ and $b = 0$ otherwise
1: $(y_1, \ldots, y_n) \leftarrow \mathsf{ArithmeticToBoolean}(q, (x_1, \ldots, x_n))$
2: **return** $\mathsf{ZeroTestBoolLog}(k, (y_1, \ldots, y_n))$

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 [CGMZ21]. We can also obtain an improved $\mathcal{O}(n^2 \cdot \log k)$ complexity using the improved arithmetic to Boolean conversion from [CGTV15]. The technique actually works for arithmetic masking modulo any integer q, since we can use [BBE+18,SPOG19] to convert from arithmetic modulo q to Boolean masking, with complexity $\mathcal{O}(n^2 \cdot \log \log q)$. In the second step, one can use the improved algorithm ZeroTestBoolLog from Appendix B.3 with complexity $\mathcal{O}(n^2 \cdot \log k)$. Therefore the overall complexity is $\mathcal{O}(n^2 \cdot \log k)$, where $k = \lceil \log_2 q \rceil$, with a number of operations:

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

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

Theorem 2. *The* ZeroTestAB *algorithm is* $(n-1)$*-NI, when* b *is given to the simulator.*

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

In Appendix B.4, we describe an alternative zero-testing from arithmetic masking based on the generic table recomputation approach from [CGMZ21]. Moreover, for $q = 2^k$ and small k, we can use the register optimization from [CGMZ21]. 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 $k = 8$.

3.3 Zero testing modulo a prime q via exponentiation

Our new technique works for prime q only. It consists in computing

$$b = 1 - x^{q-1} \pmod q \tag{1}$$

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 $x = x_1 + \cdots + x_n \pmod q$, the exponentiation $x^{q-1} \bmod q$ 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 $b = b_1 + \cdots + b_n \pmod q$, 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 B.5 the secure multiplication algorithm SecMult, already considered in [SPOG19]. We then provide in Appendix B.6 the pseudo-code of the ZeroTestExpo algorithm computing the bit b as in (1). We provide the proof of the following theorem in Appendix B.7.

Theorem 3. *The* ZeroTestExpo *algorithm is* $(n-1)$*-NI, when the output* b *is given to the simulator.*

3.4 Zero testing modulo a prime q via multiplicative masking

Our second technique also 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 to distinguish the two cases, without leaking more information about x.

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

$$u_1 \cdot x = u_1 \cdot x_1 + \cdots + u_1 \cdot x_n \pmod q$$

by computing the corresponding shares $x'_i = u_1 \cdot x_i \bmod q$ 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 and 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 $u_2, \ldots, u_n \in \mathbb{Z}_q^*$. Eventually we obtain an arithmetic sharing $(B_i)_{1 \leq i \leq n}$ satisfying:

$$u_1 \cdots u_n \cdot x = B_1 + \cdots + B_n \pmod q$$

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 $B = B_1 + \ldots + B_n \pmod q$, and we obtain:

$$u_1 \cdots u_n \cdot x = B \pmod q$$

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 ZeroTestMult algorithm taking as input the shares x_i of $x = x_1 + \cdots + x_n \pmod q$ and outputting a bit b with $b = 1$ if $x = 0$ and $b = 0$ otherwise. We provide the pseudocode of the LinearRefreshMasks algorithm in Appendix B.8.

Note that as opposed to the techniques described in the previous sections, we obtain a bit b directly in the clear. This means that when zero testing multiple coefficients at once, we can not keep an n-shared bit b and high-order combine the results of individual zero-testing, as with the previous ZeroTestBool, ZeroTestAB and ZeroTestExpo algorithms. Therefore, to test multiple coefficients at once, we will have to proceed differently (see Section 4).

Algorithm 3 ZeroTestMult

Input: $x_1, \ldots, x_n \in \mathbb{Z}_q$ for prime q.
Output: $b \in \{0,1\}$ with $b = 1$ if $\sum_i x_i = 0 \pmod q$ and $b = 0$ otherwise

$(B_1, \ldots, B_n) \leftarrow (x_1, \ldots, x_n)$
for $j = 1$ to n **do**
    $u_j \leftarrow \mathbb{Z}_q^*$
    $(B_1, \ldots, B_n) \leftarrow (u_j \cdot B_1 \bmod q, \ldots, u_j \cdot B_n \bmod q)$
    $(B_1, \ldots, B_n) \leftarrow \mathsf{LinearRefreshMasks}(q, B_1, \ldots, B_n)$
end for
$B \leftarrow B_1 + \cdots + B_n \bmod q$
if $B = 0$ **then return** 1
else return 0

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

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

The technique has therefore complexity $\mathcal{O}(n^2)$ for a single coefficient. That is, as opposed to the previous techniques, 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 3.5 that for zero testing a single coefficient, the technique is much faster than the other techniques.

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 B.9.

Theorem 4 ($(n-1)$-NI 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 $|I| \leq t$.*

3.5 Comparison of zero-test algorithms

We provide 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.

zero-testing mod q	Security order t								
	1	2	3	4	5	6	8	10	12
ZeroTestAB	439	1 393	2 593	4 514	6 681	9 289	15 913	24 386	34 428
ZeroTestExpo	182	474	900	1 460	2 154	2 982	5 040	7 634	10 764
ZeroTestMult	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 consider the zero-testing of multiple coefficients at once. For this we extend the zero-testing techniques from Section 3 to multiple coefficients. We refer to Table 1 in Section 1 for a summary of the resulting algorithms.

For Boolean masked coefficients (PolyZeroTestBool) the extension is straightforward: we can simply keep the results of individual zero-testing in n-shared form (instead of recombining the shares), and high-order compute an iterated And between those results; only at the end do we recombine the shares to output a bit b. The approach is the same for zero testing multiple coefficients arithmetically masked modulo 2^k, when using arithmetic to Boolean conversion (PolyZeroTestAB).

When working modulo a prime q, it is very advantageous to first apply the technique from [BDH+21] that reduces the zero-testing of ℓ coefficients to the zero-testing of $\kappa \ll \ell$ coefficients, with $\kappa = \lceil \lambda / \log_2 q \rceil$, where λ is the security parameter, via random linear combinations. Namely the coefficients of the linear combinations can be computed in the clear, and the complexity of this first step is only $\mathcal{O}(n)$ instead of $\mathcal{O}(n^2)$.

The remaining κ coefficients must then be zero tested all at once. For this one can use either the zero-testing based on exponentiation (ZeroTestExpo), or the zero-testing based on multiplicative masking (ZeroTestMult). When using the ZeroTestExpo algorithm, as previously we keep the resulting bit of each individual zero test in shared form. The only difference is that these bits are arithmetically masked modulo q, so we can combine them by high-order multiplication with the SecMult algorithm (instead of SecAnd as with Boolean shares). 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+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. If all coefficients are 0, the linear combination will be 0, and the algorithm will return $b = 1$ as required. If at least one of the coefficients is non-zero, the linear combination will be non-zero and the algorithm will return $b = 0$, except with error probability $1/q$. As previously, by repeating the procedure κ 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 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 ℓ coefficients:

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

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 3.1, 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:

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

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 SecAnds 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 4 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 $\oplus_i x_i^{(j)} = 0$ for all $1 \leq j \leq \ell$, and $b = 0$ otherwise

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

The number of operations is:

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

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 5. *The* PolyZeroTestBool *algorithm is* $(n-1)$*-NI, when* b *is given to the simulator.*

4.2 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 ℓ coefficients:

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

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 $\mathcal{O}(n^2 \cdot \log k)$ for $k = \lceil \log_2 q \rceil$. Therefore the total complexity is $\mathcal{O}(\ell \cdot n^2 \cdot$

$\log k$). We provide the pseudocode of the corresponding algorithm PolyZeroTestAB in Appendix C.1, with a more precise operation count. 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 6. *The* PolyZeroTestAB *algorithm is* $(n-1)$*-NI, when* b *is given to the simulator.*

4.3 Polynomial comparison modulo prime q: reduction step

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

$$y^{(k)} = \sum_{j=1}^{\ell} a_{kj} \cdot x^{(j)} \bmod q \tag{2}$$

for $1 \le k \le \kappa$, 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 5 PolyZeroTestRed [BDH+21]

Input: $q \in \mathbb{Z}$, a parameter κ and $(x_i^{(j)}) \in \mathbb{Z}_q$ for $1 \le i \le n$ and $1 \le j \le \ell$.
Output: $(y_i^{(k)}) \in \mathbb{Z}_q$ for $1 \le i \le n$ and $1 \le k \le \kappa$.

for $k = 1$ to κ **do**
  for $i = 1$ to n **do** $y_i^{(k)} \leftarrow 0$
  for $j = 1$ to ℓ **do**
    $a_{kj} \leftarrow \mathbb{Z}_q$
    for $i = 1$ to n **do** $y_i^{(k)} \leftarrow y_i^{(k)} + a_{kj} \cdot x_i^{(j)}$
  end for
end for
return $(y_i^{(k)})_{1 \le k \le \kappa,\ 1 \le i \le n}$

Now if $x^{(j)} = 0$ for all $1 \le j \le \ell$, then $y^{(k)} = 0$ for all $1 \le k \le \kappa$. If $x^{(j)} \neq 0$ for some $1 \le j \le \ell$, then for each $1 \le k \le \kappa$, we have $y^{(k)} \neq 0$, except with probability $1/q$. Therefore we must have $y^{(k)} \neq 0$ for some $1 \le k \le \kappa$, except with error probability $q^{-\kappa}$. We have therefore reduced the zero-testing of ℓ coefficients to the zero-testing of $\kappa \ll \ell$ coefficients. To reach an error probability $\le 2^{-\lambda}$ for security parameter λ, 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 (2) 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 κ coefficients $y^{(k)}$ must be zero-tested all at once. For this we describe in the next sections two efficient techniques.

This reduction technique is quite efficient because the random coefficients a_{kj} in (2) 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:

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

Theorem 7 ([BDH+21]). *The* PolyZeroTestRed *algorithm is* $(n-1)$*-NI.*

4.4 Polynomial comparison modulo q via exponentiation

The technique works modulo a prime q. As previously, we are given as input ℓ coefficients $x^{(j)} \in \mathbb{Z}_q$ with arithmetic shares $x_i^{(j)}$ modulo 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 PolyZeroTestRed described in the previous section, so there remains only κ coefficients $y^{(j)}$ to be zero-tested, from their arithmetic shares $y_i^{(j)}$ modulo q, that is $y^{(j)} = y_1^{(j)} + \cdots + y_n^{(j)} \pmod q$ for all $1 \leq j \leq \kappa$.

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

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

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 κ 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 C.2 a pseudocode description of the corresponding PolyZeroTestExpo algorithm. The proof of the following theorem is straightforward and is therefore omitted.

Theorem 8. *The* PolyZeroTestExpo *algorithm is* $(n-1)$*-NI, when* b *is given to the simulator.*

4.5 Polynomial comparison modulo q via multiplicative masking

As explained previously, when zero testing a value x modulo q using the multiplicative masking technique (Section 3.4), we obtain the resulting bit b in the clear, so we cannot zero-test the coefficients iteratively as in the previous techniques. 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+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 coefficients $x^{(j)}$ and then a CCA attack.

As previously, we consider as input an arithmetic masking of ℓ coefficients $x^{(j)}$, that is $x^{(j)} = x_1^{(j)} + \cdots + x_n^{(j)} \pmod q$ for all $1 \leq j \leq \ell$. We first apply the reduction algorithm PolyZeroTestRed described previously. 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 $y^{(j)} = y_1^{(j)} + \cdots + y_n^{(j)} \pmod q$ 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:

$$z = \sum_{j=1}^{\kappa} a^{(j)} \cdot y^{(j)} \bmod q \tag{4}$$

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 κ repetitions with randomly generated $a^{(j)}$, the error probability becomes $q^{-\kappa}$.

Equation (4) 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 Section 4.3 for the reduction step, in which the coefficients a_{kj} in (2) were 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, since otherwise an equation over the $x^{(j)}$ could be leaked with fewer than n probes. From the high-order computation of (4), we obtain the n shares z_i of the linear combination z. We then apply the zero-test procedure from Section 3.4 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 κ times, and if we always obtain $b = 1$ from the zero-test, we output 1, otherwise we output 0. We provide in Appendix C.3 a pseudocode description of the corresponding algorithm PolyZeroTestMult.

For security level λ, the error probability must satisfy $q^{-\kappa} \leq 2^{-\lambda}$, so we can take $\kappa = \lceil \lambda / \log_2 q \rceil$ repetitions. Therefore, the complexity of the second step is $\mathcal{O}(\kappa^2 n^2)$. The total complexity is therefore $\mathcal{O}(\kappa \ell n + \kappa^2 n^2)$; see Appendix C.3 for a more precise operation count. Theorems 9 and 10 below prove the soundness and security of the algorithm respectively; we refer to Appendix C.4 and C.5 for the proofs.

Theorem 9 (Soundness). *The* PolyZeroTestMult *outputs the correct answer, except with probability at most* $q^{-\kappa}$.

Theorem 10. *The* PolyZeroTestMult *algorithm is* $(n-1)$*-NI, when* b *is given to the simulator.*

4.6 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 ℓ, the asymptotic complexity of PolyZeroTestExpo and PolyZeroTestMult is $\mathcal{O}(\ell \cdot n)$, instead of $\mathcal{O}(\ell \cdot n^2)$ for PolyZeroTestAB. 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	1 329	2 045	2 826	3 685	4 934	6 262
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	1 517	1 972	2 425	3 211	4 125
PolyZeroTestExpo	79	96	116	153	185	224	254	280	345
PolyZeroTestMult	83	97	121	164	194	242	268	317	403

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	Security order t								
mod q	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
PolyZeroTestExpo	8 856	9 119	9 429	9 826	10 408	10 936	11 555	12 215	13 601
PolyZeroTestMult	8 734	9 020	9 310	9 881	10 739	11 599	12 461	13 601	15 033

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+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 the ring $R_q = \mathbb{Z}_q[X]/(X^N + 1)$ 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:

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

and are decompressed using the function $\mathsf{Decompress}_{q,d}(c) := \lfloor (q/2^d) \cdot c \rceil$, 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).

In the IND-CCA decryption 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 $\mathsf{Compress}_{q,d}$ 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 $c = \mathsf{Compress}_{q,d}(x)$. We must therefore perform the comparison with the input ciphertext $\tilde{c}$ modulo 2^d:

$$\tilde{c} \stackrel{?}{=} \mathsf{Compress}_{q,d}(x) \pmod{2^d} \tag{5}$$

There are two possible approaches to perform this comparison. The first approach consists in performing the comparison as in (5). Since the re-encrypted coefficient x is arithmetically masked modulo q, we show how to high-order compute $\mathsf{Compress}_{q,d}(x)$ 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 technique from Section 4.1. We describe in Section 5.1 the high-order computation of the Compress function.

A second approach is to avoid the computation of the Compress function, as already used in [BGR+21]. Namely instead of performing the comparison over $\{0,1\}^d$ as in (5), one can equivalently compute the set of candidates $\tilde{x}_i$ such that $\tilde{c} = \mathsf{Compress}_{q,d}(\tilde{x}_i)$. 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+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 5.4).

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 with 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 5.6.

	N	k	q	η_1	η_2	(d_u, d_v)	δ
Kyber512	256	2	3329	3	2	(10,4)	2^{-139}
Kyber768	256	3	3329	2	2	(10,4)	2^{-164}
Kyber1024	256	4	3329	2	2	(11,5)	2^{-174}

Table 7. Parameter sets for Kyber.

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+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:

$$\mathsf{Compress}_{q,d}(x) = \left\lfloor \frac{2^d \cdot x}{q} \right\rceil \bmod 2^d$$

We are given as input an arithmetic sharing of $x = x_1 + \ldots + x_n \pmod{q}$ and we want to compute a Boolean sharing of $y = \mathsf{Compress}_{q,d}(x) = y_1 \oplus \cdots \oplus y_n \in \{0,1\}^d$. We stress that in

[BGR+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+21] and [CGMZ21].

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:

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

The rounding can be computed by writing:

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

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 $z_1, \ldots, z_n$, followed by a logical shift by α bits. This can be done with complexity $\mathcal{O}((d+\alpha) \cdot n^2)$ using [CGV14]. By definition we obtain:

$$y_1 \oplus \cdots \oplus y_n = \left\lfloor \left(\sum_{i=1}^{n} z_i \right) / 2^{\alpha} \right\rfloor \pmod{2^d} \tag{6}$$

and eventually we output the Boolean shares $y_1, \ldots, y_n$. We show below that we indeed have $\mathsf{Compress}_{q,d}(x) = y_1 \oplus \cdots \oplus y_n$ as required, under the condition $2^\alpha > q \cdot n$. This condition determines the number α of bits of precision as a function of the number of shares n. We provide the pseudocode in Algorithm 6 below.

Algorithm 6 HOCompress

Input: $x_1, \ldots, x_n \in \mathbb{Z}_q$
Output: $y_1, \ldots, y_n \in \{0,1\}^d$ such that $y_1 \oplus \cdots \oplus y_n = \mathsf{Compress}_{q,d}(x_1 + \cdots + x_n)$
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: $(c_1, \ldots, c_n) \leftarrow \mathsf{ArithmeticToBoolean}(d+\alpha, (z_1, \ldots, z_n))$
5: **for** $i = 1$ to n **do** $y_i \leftarrow c_i \gg \alpha$
6: **return** $y_1, \ldots, y_n$

Theorem 11 (Soundness). *Given $x_1, \ldots, x_n \in \mathbb{Z}_q$ as input for odd $q \in \mathbb{N}$, the algorithm* HOCompress *computes $y_1, \ldots, y_n \in \{0,1\}^d$ such that $y_1 \oplus \cdots \oplus y_n = f(x_1 + \cdots + x_n)$ where $f(x) = \lfloor x \cdot 2^d/q \rceil \bmod 2^d$.*

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

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

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 $|e| \leq n/2$:

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

From (6), if we ensure that $0 \leq \delta/(2q) + e \cdot 2^{-\alpha} < 1$, then we must have $y = f(x) = y_1 \oplus \cdots \oplus y_n$ 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 $2^{\alpha} > q \cdot n$. □

Complexity of HOCompress. The number of operations of the HOCompress algorithm above is:

$$T_{\mathsf{HOComp}}(n, d, q) = 5n + 1 + T_{\mathsf{AB}}(d + \alpha, n)$$

We refer to [CGV14] for the operation count of arithmetic to Boolean conversion, with $T_{\mathsf{AB}}(d + \alpha, n) = \mathcal{O}((d+\alpha) \cdot n^2)$. 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 12 ($(n-1)$-NI security). *The* HOCompress *algorithm achieves the $(n-1)$-NI property.*

Polynomial comparison with Compress. Recall that we must perform the comparison $\tilde{c} \stackrel{?}{=} \mathsf{Compress}_{q,d}(x)$, where for simplicity we consider a single coefficient $\tilde{c}$. By applying the HOCompress algorithm, we obtain n Boolean shares such that $c = c_1 \oplus \cdots \oplus c_n$. We must therefore zero-test the value $(c_1 \oplus \tilde{c}) \oplus c_2 \oplus \cdots \oplus c_n$, which can be done using the ZeroTestBool algorithm from Section 3.1.

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 Section 4.1 to perform the comparison.

5.2 Polynomial comparison for Kyber without Compress

In this section we describe an alternative technique for ciphertext comparison, already used in [BGR+21], that performs the comparison on uncompressed ciphertexts, and avoids the high-order computation of the $\mathsf{Compress}_{q,d}(x)$ function. As previously, we first consider for simplicity

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

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

Given $\tilde{c}$ as input, we must therefore compute the list of candidates $\mathsf{Compress}_{q,d}^{-1}(\tilde{c})$, and check whether x belongs to the set of candidates. Given $0 \leq a < b < q$, we denote by $[a, b]_q$ the discrete interval $\{a, a+1, \ldots, b\}$; similarly given $0 \leq b < a < q$, we denote by $[a, b]_q$ the discrete interval $\{a, a+1, \ldots, q-1, 0, 1, \ldots, b\}$. In the following, we show that we always have $\mathsf{Compress}_{q,d}^{-1}(\tilde{c}) = [a, b]_q$ for some a, b.

We can then distinguish two cases. If the number of candidates is small, we can perform individual comparisons. More precisely, letting $\{\tilde{x}_1, \ldots, \tilde{x}_m\} = \mathsf{Compress}_{q,d}^{-1}(\tilde{c})$ 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. Alternatively, for a large number of candidates, the authors of [BGR+21] describe a high-order algorithm checking that $x \in [a, b]_q$ by performing two high-order comparisons. We describe the two methods in more details in the next subsections.

5.3 Computing the set of candidates

Given a compressed coefficient $\tilde{c}$, we must compute the list of candidates $\mathsf{Compress}_{q,d}^{-1}(\tilde{c})$. From [BDK+18], we know that for any $x \in \mathbb{Z}_q$ such that $\tilde{c} = \mathsf{Compress}_{q,d}(x)$, letting the value $y = \mathsf{Decompress}_{q,d}(\tilde{c})$ we must have:

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

Therefore the number of candidates is upper-bounded by $2B_{q,d} + 1$. 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 to test with $\mathsf{Compress}$. We provide the proof in Appendix D.1.

Lemma 3. *Assume $d < \lceil \log_2 q \rceil$. Let $\tilde{c} \in \mathbb{Z}_{2^d}$ and let $y = \mathsf{Decompress}_{q,d}(\tilde{c})$. We have $[y - B_{q,d} + 1, y + B_{q,d} - 1]_q \subset \mathsf{Compress}_{q,d}^{-1}(\tilde{c}) \subset [y - B_{q,d}, y + B_{q,d}]_q$.*

Generating the list of candidates. From the above lemma, to generate the list of candidates $\mathsf{Compress}_{q,d}^{-1}(\tilde{c})$, 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 $\mathsf{Compress}_{q,d}(a) = c$ and $\mathsf{Compress}_{q,d}(b) = c$. We provide the pseudocode in Algorithm 7 below.

Algorithm 7 CompressInv

Input: $\tilde{c} \in \mathbb{Z}_{2^k}$
Output: $a, b \in \mathbb{Z}$ such that $\mathsf{Compress}_{q,d}^{-1}(\tilde{c}) = [a, b]_q$.

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

Number of candidates. We provide in Table 8 the value of the upper-bound $2B_{q,d}+1$ on the number of candidates, and the maximum number of candidates N_{max}, for $q = 3329$. We see that individual comparisons are feasible only for $d = 10, 11$, while we must use range comparison for $d = 4, 5$. We describe the two techniques in the next section.

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

Table 8. Upper-bound on the number of candidates, for $q = 3329$.

5.4 Individual comparison

Letting $\{\tilde{x}_1, \ldots, \tilde{x}_m\} = \mathsf{Compress}_{q,d}^{-1}(\tilde{c})$ 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 $x = x_1 + \cdots + x_n \pmod q$, we high-order compute the value

$$z = \prod_{i=1}^{m} (x - \tilde{x}_i) \bmod q \tag{7}$$

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 D.2 the pseudocode description of the SecMultList algorithm, computing the n shares of z in (7), from the input shares x_i of x. For m candidates, the number of operations for high-order computing z is therefore at most:

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

As a second step, one can apply a high-order zero-test of z modulo q, either the ZeroTestExpo algorithm from Section 3.3, or the ZeroTestMult algorithm from Section 3.4.

The above applies for a single coefficient x. In reality we must compare ℓ 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 $z^{(j)}$. Then a polynomial zero-test is applied to the set of arithmetically masked $z^{(j)}$'s modulo q, either the PolyZeroTestExpo algorithm from Section 4.4, or the PolyZeroTestMult algorithm from Section 4.5.

5.5 Polynomial comparison with range test [BGR+21]

When the number of candidates in $\mathsf{Compress}_{q,d}^{-1}(\tilde{c})$ is too large (which is the case for $d = 4, 5$), we cannot perform individual comparisons as in the previous section. Instead, we must test whether $x \in [a, b]_q = \mathsf{Compress}_{q,d}^{-1}(\tilde{c})$ by performing two high-order comparisons with the interval bounds a and b. We recall the technique from [BGR+21] in Appendix D.3, with the pseudo-code of the RangeTestShares algorithm and the operation count.

5.6 Ciphertext comparison in Kyber: hybrid approach

We first compare in Table 9 the efficiency of the approaches with and without Compress. For the coefficients with compression to $d_u = 10$ bits, without using the Compress function, since the

number of candidates is small, we can use either the RangeTestShares algorithm from [BGR+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 HOCompress 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+21]. But we see that our HOCompress is nevertheless faster. Namely, it uses only a single arithmetic to Boolean conversion with a power-of-two modulus, whereas RangeTestShares uses two arithmetic to Boolean conversions modulo q, which is more costly than with a power-of-two modulus.

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 HOCompress 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	8	9
$d_u = 10$	RangeTestShares [BGR+21]	707	2 318	4 314	7 577	11 225	15 620	20 400	26 809	33 603
	SecMultList	45	120	230	375	555	770	1 020	1 305	1 625
	HOCompress	131	868	1 579	3 181	4 898	6 764	8 809	11 823	15 611
$d_v = 4$	RangeTestShares [BGR+21]	707	2 318	4 314	7 577	11 225	15 620	20 400	26 809	33 603
	HOCompress	101	658	1 195	2 431	3 740	5 162	6 721	9 015	12 041

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

Procedure for ciphertext comparison. 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 5.4 (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 HOCompress algorithm with $d_v = 4$ bits. We obtain a set of ℓ_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 ℓ_2 coefficients of the input ciphertext $\tilde{c}$ to the first Boolean share of each of the corresponding ℓ_2 coefficients $c^{(j)}$. This gives a vector of ℓ_2 coefficients $x^{(j)}$ for $1 \leq j \leq \ell$, which are Boolean masked with n shares, and that must all be equal to 0.
4. We apply the PolyZeroTestBool algorithm (Alg. 4) to the set of ℓ_2 coefficients $x^{(j)}$, but without recombining the shares at the end of the ZeroTestBoolLog algorithm. That is, we obtain Boolean shares b_i for $1 \leq i \leq n$, with $b' = b_1 \oplus \cdots \oplus b_n$ and $b' = 1$ if the ℓ_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. 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.

The number of operations is then:

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

5.7 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+21] without Compress, and then our hybrid approach with the PolyZeroTestExpo and 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. We also provide in Table 12 the number of 32-bit random values.

Polynomial comparison in Kyber	Security order t								
	1	2	3	4	5	6	7	8	9
Without Compress [BGR+21]	786	2 574	4 792	8 413	12 465	17 346	22 657	29 770	37 313
Hybrid, with PolyZeroTestExpo	121	351	606	1 071	1 584	2 156	2 794	3 650	4 724
Hybrid, with PolyZeroTestMult	121	350	603	1 065	1 575	2 144	2 778	3 629	4 697

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+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.

Polynomial comparison in Kyber	Security order t							
	1	2	3	4	5	6	7	8
Without Compress [BGR+21]	79	309	618	1 146	1 755	2 512	3 350	4 476
Hybrid, with PolyZeroTestExpo	12	31	50	94	145	202	265	356
Hybrid, with PolyZeroTestMult	12	31	50	94	145	202	264	354

Table 12. Number of calls to the rand() function (outputting a 32-bit value), in thousands of calls, rounded down to the closest thousand, for the three methods considered in Table 10.

6 Fully masked implementation of Kyber

Kyber is a lattice-based encryption scheme and a finalist of the third round of the NIST competition [BDK+18,ABD+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 level 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 η_1, d_t, d_u and d_v (see Table 7).

In the following, we start with an overview of ring-LWE encryption [LPR10], and then 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)

Ring-LWE IND-CPA encryption. 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, for some $N \in \mathbb{Z}$ and an integer q. Let $a \in R_q$ be a public random polynomial. Let χ 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

$$\begin{aligned} c_1 &= a \cdot e_1 + e_2 \\ c_2 &= t \cdot e_1 + e_3 + \lfloor q/2 \rceil \cdot m \end{aligned} \tag{8}$$

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:

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

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 $\lfloor q/2 \rceil$, and to 1 otherwise. More precisely, we decode the message m as $m = \mathsf{th}(c_2 - s \cdot c_1)$, where th applies coefficient-wise the threshold function $\mathsf{th} : \mathbb{Z}_q \to \{0, 1\}$:

$$\mathsf{th}(x) = \begin{cases} 0 & \text{if } x \in (0, q/4) \cup (3q/4, q) \\ 1 & \text{if } x \in (q/4, 3q/4) \end{cases} \tag{9}$$

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 χ_η as the centered binomial distribution with support $\{-\eta, \ldots, \eta\}$, and extended to the distribution of polynomials of degree N with entries independently sampled from χ_η. 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 η_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 $c_2 = \mathbf{t}^T \cdot \mathbf{r} + e_2 + \lfloor q/2 \rceil \cdot m$, 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 η_2. To decrypt a ciphertext $(\mathbf{c}_1, c_2)$, one computes as previously:

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

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+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 8 Kyber.CPA.KeyGen()

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

Algorithm 9 Kyber.CPA.Dec$(sk, c = (\mathbf{u}, v))$

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

Algorithm 10 Kyber.CPA.Enc(pk, m)

1: $r \leftarrow \{0,1\}^{256}$
2: $\mathbf{t} := \mathsf{Decompress}_{q,d_t}(\mathbf{t})$
3: $\mathbf{A} \leftarrow \mathcal{R}_q^{k \times k} := \mathsf{Sam}(\rho)$
4: $\mathbf{r}, \mathbf{e}_1, e_2 \leftarrow \chi_{\eta_1}^k \times \chi_{\eta_2}^k \times \chi_{\eta_2} := \mathsf{Sam}(r)$
5: $\mathbf{u} := \mathsf{Compress}_{q,d_u}(\mathbf{A}^T\mathbf{r} + \mathbf{e}_1)$
6: $v := \mathsf{Compress}_{q,d_v}(\mathbf{t}^T\mathbf{r} + e_2 + \lfloor q/2 \rceil \cdot m)$
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+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 12 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 γ-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+21]), because for the simulation of the probes the bit b can be given for free to the simulator.

Algorithm 11 Kyber.Encaps(pk)

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

Algorithm 12 Kyber.Decaps($sk = (\mathbf{s}, z, \mathbf{t}, \rho), c = (\mathbf{u}, v)$)

1: $m' := \mathsf{Kyber.CPA.Dec}(\mathbf{s}, c)$
2: $(\hat{K}', r') := G(H(pk), m')$
3: $(\mathbf{u}', v') := \mathsf{Kyber.CPA.Enc}((\mathbf{t}, \rho), m'; r')$
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 12), using the techniques from the previous sections.

1. We consider Line 1 of Algorithm 12, 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 $\mathbf{s} = \mathbf{s}_1 + \cdots + \mathbf{s}_n \pmod q$, 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 $\mathsf{Compress}_{q,1}$ function on this value, which is the same as the threshold function th from (9). For this we use the modulus switching and table recomputation technique from [CGMZ21], which outputs a Boolean masked message $m' = m_1 \oplus \cdots \oplus m_n = \mathsf{th}(v - \mathbf{s}^T\mathbf{u})$.
2. At Line 2 of Algorithm 12, 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 12, we start with Line 4 of Algorithm 10 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 [CGMZ21], 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 10. We obtain the values $\mathbf{A}^T \cdot \mathbf{r} + \mathbf{e}_1$ and $\mathbf{t}^T \cdot \mathbf{r} + e_2 + \lfloor q/2 \rceil \cdot m$ arithmetically n-shared modulo q. In particular, the n-shared value $\lfloor q/2 \rceil \cdot m$ is obtained using the table-based Boolean to arithmetic modulo q conversion from [CGMZ21].
5. At Line 6 of Algorithm 10, the n-shared value $\mathbf{t}^T \cdot \mathbf{r} + e_2 + \lfloor q/2 \rceil \cdot m$ is high-order compressed into v' using the HOCompress 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 12, we use the hybrid technique from Section 5.6 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.

We describe in Section 8 the implementation results of the fully masked Kyber.Decaps.

7 Fully masked implementation of Saber

7.1 The Saber Key Encapsulation Mechanism (KEM)

Saber [BMD+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 13. 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 13 Saber.CPA.KeyGen()

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

	N	k	q	p	T	μ
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 13. Parameter set for Saber.

Algorithm 14 Saber.CPA.Enc(pk, m)

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

Algorithm 15 Saber.CPA.Dec$(sk, c = (\mathbf{u}, c_m))$

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

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

Algorithm 16 Saber.Encaps(pk)

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

Alg. 17 Saber.Decaps$(sk = (\mathbf{s}, z, \mathbf{t}, \rho), c)$

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

7.2 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 17), using the techniques from the previous sections.

1. We consider Line 1 of Algorithm 17. 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 17, 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 17, we start with Line 3 of Algorithm 14 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 [CGMZ21], based on Boolean to arithmetic modulo q conversion (based on table recomputation).
4. We proceed with Line 4 of Algorithm 14. 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 14 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 17, 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 5.6 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.

8 Practical implementation

We have implemented in C a high-order version of the Kyber.Decaps and Saber.Decaps algorithms, following the description of sections 6.2 and 7.2 respectively. For both schemes, we have targeted the parameter set corresponding to NIST security category 3 (parameters Kyber768 and Saber, see tables 7 and 13). We have run our implementation on a laptop. We provide the source code of the laptop implementation at:

https://github.com/fragerar/HOTableConv/tree/main/Masked_KEMs

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.

8.1 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 18 (see Appendix A), starting from a 32-bit random number generator. The timings are summarized in Table 14.

	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

Table 14. Kyber.Decaps cycles counts on Intel(R) Core(TM) i7-1065G7, in thousands of cycles.

8.2 Saber

Our high-order implementation of Saber.Decaps follows the description from Section 7.2. The timings are summarized in Table 15.

	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

Table 15. Saber.Decaps cycles counts on Intel(R) Core(TM) i7-1065G7, in thousands of cycles.

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 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 and Saber schemes at any order, with the practical results of a C implementation.

References

ABD+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+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+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.

BDH+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. Cryptology ePrint Archive, Report 2021/104, 2021. https://eprint.iacr.org/2021/104.

BDK+18. Joppe W. Bos, Léo Ducas, Eike Kiltz, Tancrède 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+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+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+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.

CGMZ21. Jean-Sebastien Coron, François Gerard, Simon Montoya, and Rina Zeitoun. High-order table-based conversion alg. and masking lattice-based encryption, 2021.

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.

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.

FBR+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 *Advances in Cryptology - CRYPTO '99, 19th Annual International Cryptology Conference, Santa Barbara, California, USA, August 15-19, 1999, 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 *Theory of Cryptography - 15th International Conference, TCC 2017, Baltimore, MD, USA, November 12-15, 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 *Advances in Cryptology - CRYPTO 2003, 23rd Annual International Cryptology Conference, Santa Barbara, California, USA, August 17-21, 2003, Proceedings*, pages 463–481, 2003.

LPR10. Vadim Lyubashevsky, Chris Peikert, and Oded Regev. On ideal lattices and learning with errors over rings. In *Advances in Cryptology - EUROCRYPT 2010, 29th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Monaco / French Riviera, May 30 - June 3, 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.

MAA+20. Dustin Moody, Gorjan Alagic, Daniel Apon, David Cooper, Quynh Dang, John Kelsey, Yi-Kai Liu, Carl Miller, Rene Peralta, Ray Perlner, Angela Robinson, Daniel Smith-Tone, and Jacob Alperin-Sheriff. Status report on the second round of the NIST post-quantum cryptography standardization process, 2020-07-22 2020.

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 *Cryptographic Hardware and Embedded Systems - CHES 2017 - 19th International Conference, Taipei, Taiwan, September 25-28, 2017, Proceedings*, pages 513–533, 2017.

RP10. Matthieu Rivain and Emmanuel Prouff. Provably secure higher-order masking of AES. In *Cryptographic Hardware and Embedded Systems, CHES 2010, 12th International Workshop, Santa Barbara, CA, USA, August 17-20, 2010. Proceedings*, pages 413–427, 2010.

SPOG19. Tobias Schneider, Clara Paglialonga, Tobias Oder, and Tim Güneysu. Efficiently masking binomial sampling at arbitrary orders for lattice-based crypto. In *Public-Key Cryptography - PKC 2019 - 22nd IACR International Conference on Practice and Theory of Public-Key Cryptography, Beijing, China, April 14-17, 2019, Proceedings, Part II*, 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 18 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: $\quad r \leftarrow \{0,1\}^k$
4: **end while**
5: $r := r \bmod q^2$
6: **return** $(r \bmod q, \lfloor r/q \rfloor)$

B Zero testing a single value

B.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.

Algorithm 19 SecAnd

Input: $k \in \mathbb{N}$, $x_1, \ldots, x_n \in \{0,1\}^k$, $y_1, \ldots, y_n \in \{0,1\}^k$

Output: $z_1, \ldots, z_n \in \{0,1\}^k$, 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** $z_1, \ldots, z_n$

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

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

Lemma 4 ([BBD+16]). *The* SecAnd *algorithm is* $(n-1)$-SNI.

B.2 Mask refreshing

We recall the RefreshMasks algorithm, where the operations are performed in any finite field $\mathbb{F}$, including $\mathbb{Z}_q$ for prime q.

Algorithm 20 RefreshMasks

Input: $a_1, \ldots, a_n$

Output: $c_1, \ldots, c_n$ 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: $r \leftarrow \mathbb{F}$, $c_i \leftarrow c_i + r$, $c_j \leftarrow c_j - r$
5: **end for**
6: **end for**
7: **return** $c_1, \ldots, c_n$

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

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

Lemma 5 ([BBD+16]). *The* RefreshMasks *algorithm is* $(n-1)$-SNI.

B.3 Boolean zero-test with complexity $\mathcal{O}(n^2 \cdot \log k)$

Let $x \in \{0,1\}^k$ and let $x = x_1 \oplus \cdots \oplus x_n$ 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

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

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:

$$x^{(k-1)} \wedge \cdots \wedge x^{(0)} = \mathsf{LSB}\left((f_{m-1} \circ \cdots \circ f_0)(x)\right) \tag{10}$$

Therefore to zero-test x, we can compute:

$$\overline{x^{(k-1)} \vee \cdots \vee x^{(0)}} = \mathsf{LSB}\left((f_{m-1} \circ \cdots \circ f_0)(\bar{x})\right)$$

We describe in Algorithm 21 below the high-order computation of the previous equation with n shares, using the SecAnd and RefreshMasks algorithms from sections B.1 and B.2.

Algorithm 21 ZeroTestBoolLog

Input: $k \in \mathbb{Z}$ and $x_1, \ldots, x_n \in \{0,1\}^k$
Output: $b \in \{0,1\}$ with $b = 1$ if $\oplus_{i=1}^n x_i = 0$ 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: $\quad (z_1, \ldots, z_n) \leftarrow \mathsf{RefreshMasks}(y_1 \gg 2^i, \ldots, y_n \gg 2^i)$
6: $\quad (y_1, \ldots, y_n) \leftarrow \mathsf{SecAnd}(m, (y_1, \ldots, y_n), (z_1, \ldots, z_n))$
7: **end for**
8: $(b_1, \ldots, b_n) \leftarrow \mathsf{RefreshMasks}(y_1 \,\&\, 1, \ldots, y_n \,\&\, 1)$
9: **return** $b_1 \oplus \cdots \oplus b_n$

Theorem 13 (Soundness). *Given as input* $x_1, \ldots, x_n \in \{0,1\}^k$*, the* ZeroTestBoolLog *algorithm outputs* $b = 1$ *if* $\oplus_{i=1}^n x_i = 0$ *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 $(f_{m-1} \circ \cdots \circ f_0)(y)$ where $f_i(y) = y \wedge (y \gg 2^i)$. To ease reading, we denote by $F_i(y)$ the value $(f_i \circ \cdots \circ f_0)(y)$. In the following we prove by recurrence on i for $i < m$ that the j-th bit of $F_i(y)$ is

$$(F_i(y))^{(j)} = y^{(j+2^{i+1}-1)} \wedge \cdots \wedge y^{(j)} \ , \tag{11}$$

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

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

Indeed, we have

$$(F_{i+1}(y))^{(j)} = (f_{i+1}(F_i(y)))^{(j)} = (F_i(y) \wedge ((F_i(y) \gg 2^{i+1})))^{(j)} = (F_i(y))^{(j)} \wedge (F_i(y))^{(j+2^{i+1})} \ .$$

By using the recurrence hypothesis in Equation (11), we get

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

which terminates the recursive proof.

In particular, for $i = m - 1$, we can use Equation (11) 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:

$$(F_{m-1}(y))^{(0)} = \mathsf{LSB}\left((f_{m-1} \circ \cdots \circ f_0)(y)\right) = y^{(2^m-1)} \wedge \cdots \wedge y^{(0)} = y^{(k-1)} \wedge \cdots \wedge y^{(0)} \ ,$$

as specified in equation (10). 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 $y = \overline{x}$, we obtain as required:

$$\mathsf{LSB}\left((f_{m-1} \circ \cdots \circ f_0)(\bar{x})\right) = \overline{\overline{y^{(k-1)}} \vee \cdots \vee \overline{y^{(0)}}} = \overline{x^{(k-1)} \vee \cdots \vee x^{(0)}} \ .$$

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

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

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

Theorem 14. *The* ZeroTestBoolLog *algorithm is* $(n-1)$-NI*, when* b *is given to the simulator.*

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 thanks to the last RefreshMasks, the full algorithm is $(n-1)$-NI when b is given to the simulator.

□

B.4 Zero testing modulo 2^k for small k via table recomputation

The technique is a direct application of the table-based conversion algorithm from [CGMZ21]. Namely the table recomputation from [CGMZ21] can high-order compute any function $f : G \to H$, 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 [CGMZ21]. 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:

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

Given as input the shares x_i with $x = x_1 + \cdots + x_n \pmod q$, 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 $x_1 + \cdots + x_{n-1} \bmod q$, 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 $\mathsf{ROR}[a](R)$ the cyclic rotation of a register R by a bits to the right.

Algorithm 22 ZeroTestTable

Input: $x_1, \ldots, x_n \in \mathbb{Z}_{2^k}$
Output: $b \in \{0, 1\}$, with $b = 1$ if $x_1 + \cdots + x_n = 0 \pmod{2^k}$, 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** $R_j \leftarrow \mathsf{ROR}[x_i](R_j)$
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: $(b_1, \ldots, b_n) \leftarrow \mathsf{RefreshMasks}_{\{0,1\}}(R_1[x_n], \ldots, R_n[x_n])$
11: **return** $b_1 \oplus \cdots \oplus b_n$

Theorem 15. *The* ZeroTestTable *algorithm is* $(n-1)$*-NI, when* b *is given to the simulator.*

Proof. From [CGMZ21], the ZeroTestTable algorithm up to Line 10 is $(n-1)-\mathsf{SNI}$. Thanks to the last RefreshMasks, it is actually free-$(n-1)-\mathsf{SNI}$. Therefore the full algorithm is $(n-1)-\mathsf{NI}$ when the output b is given to the simulator. □

B.5 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 23 SecMult

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

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

B.6 Zero testing modulo a prime q via exponentiation

Algorithm 24 SecExpo

Input: $A_1, \ldots, A_n \in \mathbb{Z}_q$ with $\sum_i A_i = x \pmod q$ for prime q, an exponent e.
Output: $B_1, \ldots, B_n \in \mathbb{Z}_q$ with $\sum_i B_i = x^e \pmod q$

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

Algorithm 25 ZeroTestExpoShares

Input: $A_1, \ldots, A_n \in \mathbb{Z}_q$ for prime q.
Output: $B_1, \ldots, B_n$ 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: $(B_1, \ldots, B_n) \leftarrow \mathsf{SecExpo}((A_1, \ldots, A_n), q-1)$
2: $(B_1, \ldots, B_n) \leftarrow (1 - B_1 \bmod q, -B_2 \bmod q, \ldots, -B_n \bmod q)$
3: **return** $(B_1, \ldots, B_n)$

Algorithm 26 ZeroTestExpo

Input: $A_1, \ldots, A_n \in \mathbb{Z}_q$ for prime q.
Output: $b \in \{0,1\}$ with $b = 1$ if $\sum_i A_i = 0 \pmod q$ and $b = 0$ otherwise
1: $(B_1, \ldots, B_n) \leftarrow \mathsf{ZeroTestExpoShares}(A_1, \ldots, A_n)$
2: $(C_1, \ldots, C_n) \leftarrow \mathsf{RefreshMasks}(B_1, \ldots, B_n)$
3: **return** $C_1 + \cdots + C_n \bmod q$

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:

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

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

B.7 Proof of Theorem 3

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.

The ZeroTestExpo algorithm is composed by the ZeroTestExpoShares algorithm which is $(n-1)$-SNI, followed by a RefreshMasks performed at Step 2 which is free-$(n-1)$-SNI from Lemma 1. Therefore, steps 1 and 2, that is, the whole ZeroTestExpo before combining the output shares $(C_1, \ldots, C_n)$, is free-$(n-1)$-SNI. One can then use Corollary 1 to deduce that the ZeroTestExpo algorithm is $(n-1)$-NI when the output $b = C_1 + \cdots + C_n \bmod q$ is given to the simulator.

B.8 The LinearRefreshMasks algorithm

Algorithm 27 LinearRefreshMasks

Input: $q \in \mathbb{Z}$ and $x_1, \ldots, x_n \in \mathbb{Z}_q$
Output: $y_1, \ldots, y_n \in \mathbb{Z}_q$ such that $y_1 + \cdots + y_n = x_1 + \cdots + x_n \pmod q$
1: $y_n \leftarrow x_n$
2: **for** $j = 1$ to $n-1$ **do**
3: $\quad r_j \leftarrow \mathbb{Z}_q$
4: $\quad y_j \leftarrow x_j + r_j \bmod q$
5: $\quad y_n \leftarrow y_n - r_j \bmod q$
6: **end for**
7: **return** $y_1, \ldots, y_n$

B.9 Proof of Theorem 4

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 $j^\star$ be the index of this non-probed loop. For all probed variables B_i between Steps 3 and 5 in loop indices $j < j^\star$, 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 $j \geq j^\star$, 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 $j^\star$, since LinearRefreshMasks has not been proved, 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 $u_{j^\star}$ 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 $j^\star$ 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 $(n-1)$-NI, when b is given to the simulator.

C Polynomial comparison

C.1 Polynomial comparison modulo 2^k via A. to B. conversion

Algorithm 28 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 ℓ **do**
2: $\quad (y_i^{(j)})_{1 \leq i \leq n} \leftarrow \mathsf{ArithmeticToBoolean}(q, (x_i^{(j)})_{1 \leq i \leq n})$
3: **end for**
4: **return** $\mathsf{PolyZeroTestBool}(k, (y_i^{(j)}))$

The number of operations is

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

C.2 Polynomial comparison modulo q via exponentiation

Algorithm 29 PolyZeroTestExpo

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

$(y_i^{(k)})_{1\le k\le \kappa,\, 1\le i\le n} \leftarrow \mathsf{PolyZeroTestRed}((x_i^{(j)})_{1\le j\le \ell,\, 1\le i\le n})$
$(B_1, \ldots, B_n) \leftarrow \mathsf{ZeroTestExpoShares}(y_1^{(1)}, \ldots, y_n^{(1)})$
for $j = 2$ to κ **do**
    $(C_1, \ldots, C_n) \leftarrow \mathsf{ZeroTestExpoShares}(y_1^{(j)}, \ldots, y_n^{(j)})$
    $(B_1, \ldots, B_n) \leftarrow \mathsf{SecMult}((B_1, \ldots, B_n), (C_1, \ldots, C_n))$
end for
$(C_1, \ldots, C_n) \leftarrow \mathsf{RefreshMasks}(B_1, \ldots, B_n)$
$B \leftarrow C_1 + \cdots + C_n \bmod q$
if $B = 1$ **then return** 1
else return 0

The number of operations is:

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

C.3 Polynomial comparison modulo q via multiplicative masking

Algorithm 30 PolyZeroTestMult

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

$(y_i^{(k)})_{1\le k\le \kappa,\, 1\le i\le n} \leftarrow \mathsf{PolyZeroTestRed}((x_i^{(j)})_{1\le j\le \ell,\, 1\le i\le n})$
$b \leftarrow 0$
for $k = 1$ to κ **do**
    for $i = 1$ to n **do** $z_i \leftarrow 0$
    for $j = 1$ to ℓ **do**
        for $i = 1$ to n **do** $a_i^{(j)} \leftarrow \mathbb{Z}_q$
        $(z_1^{(j)}, \ldots, z_n^{(j)}) \leftarrow \mathsf{SecMult}((a_1^{(j)}, \ldots, a_n^{(j)}), (y_1^{(j)}, \ldots, y_n^{(j)}))$
        for $i = 1$ to n **do** $z_i \leftarrow z_i + z_i^{(j)} \bmod q$
    end for
    $b_k \leftarrow \mathsf{ZeroTestMult}(z_1, \ldots, z_n)$
    $b \leftarrow b + b_k$
end for
if $b = \kappa$ **then** $b \leftarrow 1$ **else** $b \leftarrow 0$
return b

The complexity of the PolyZeroTestMult algorithm is

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

C.4 Proof of Theorem 9

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 κ 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 κ iterations, with at least one coefficient $y^{(j)} \neq 0$, is $(1/q)^\kappa = q^{-\kappa}$.

C.5 Proof of Theorem 10

As before, we denote by PolyZeroTestMultLoop, one loop iteration on k of the PolyZeroTestMult algorithm (line 4 to 11). We write $y^{(j)} = \sum_{i=1}^{n} y_i^{(j)} \bmod q$. 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 4. 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_0: 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_1: 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 $a^{(j)} = \sum_{i=1}^{n} a_i^{(j)} \bmod q$. 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_2: we show that we can still perform a perfect simulation as in Game_1, 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 $\mathsf{SecMult}$ and the $(n-1)$-NI property of $\mathsf{ZeroTestMult}$ 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_2 can be satisfied by generating all shares $a_i^{(j)}$ for $i \neq i^\star$ uniformly at random, and by fixing $a_{i^\star}^{(j)}$, without changing the distribution of the shares $a_i^{(j)}$, for some $i^\star \notin J$. Finally, since the knowledge of $a_{i^\star}^{(j)}$ 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 Polynomial comparison for Kyber

D.1 Proof of Lemma 3

We first show that if $x \in \mathsf{Compress}_{q,d}^{-1}(\tilde{c})$, then we must have $|x - y \bmod^{\pm} q| \leq B_{q,d}$, where $y = \mathsf{Decompress}_{q,d}(\tilde{c})$. This will imply $\mathsf{Compress}_{q,d}^{-1}(\tilde{c}) \subset [y - B_q, \ldots, y + B_q]_q$. Namely in this case we have $\tilde{c} = \mathsf{Compress}_{q,d}(x)$, and therefore we have

$$y = \mathsf{Decompress}_{q,d}(\mathsf{Compress}_{q,d}(x)) = \left\lfloor \frac{q}{2^d} \cdot \left(\left\lfloor \frac{2^d}{q} \cdot x \right\rceil \bmod 2^d \right) \right\rceil$$

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

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

Conversely, we show that if $|x - y \bmod^{\pm} q| \leq B_{q,d} - 1$, then we must have $x \in \mathsf{Compress}_{q,d}^{-1}(\tilde{c})$. This will imply $[y - B_q + 1, \ldots, y + B_q - 1]_q \subset \mathsf{Compress}_{q,d}^{-1}(\tilde{c})$. Namely we write again $\lfloor (2^d/q) \cdot x \rceil =$

$(2^d/q) \cdot x + \varepsilon$ for some $|\varepsilon| \leq 1/2$, and we can write:

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

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

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

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

D.2 The SecMultList algorithm

Algorithm 31 SecMultList

Input: $x_1, \ldots, x_n \in \mathbb{Z}_q$ s.t. $\sum_i x_i \pmod q = x$, a prime q, an index m and $a_1, \ldots a_m \in \mathbb{Z}_q$
Output: $z_1, \ldots, z_n \in \mathbb{Z}_q$ such that $\sum_{i=1}^n z_i = \prod_{i=1}^m (x - a_i) \pmod q$

1: $(z_1, z_2, \ldots, z_n) \leftarrow (1, 0, \ldots 0)$
2: **for** $i = 1$ to m **do**
3: $\quad (z_1, z_2, \ldots, z_n) \leftarrow \mathsf{SecMult}((z_1, \ldots, z_n), (x_1 - a_i \bmod q, x_2, \ldots, x_n))$
4: **end for**
5: **return** $(z_1, \ldots, z_n)$

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. □

D.3 Polynomial comparison with range test [BGR+21]

When the number of candidates in $\mathsf{Compress}_{q,d}^{-1}(\tilde{c})$ is too large, we cannot perform individual comparisons as in the previous section. Instead, we must test whether $x \in [a, b]_q = \mathsf{Compress}_{q,d}^{-1}(\tilde{c})$ by performing two high-order comparisons with the interval bounds a and b. We recall the technique from [BGR+21].

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 $b - a \bmod^{\pm} q \leq \Delta$. Recall that we have the upper-bound $b - a \bmod^{\pm} q \leq 2 \cdot B_{q,d} + 1 = 2 \cdot \left\lfloor q/2^{d+1} \right\rceil + 1$. Therefore the assumption is satisfied for Kyber for $d \geq 2$.

Taking as input the shares x_i of $x = x_1 + \cdots + x_n \pmod q$, 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:

$$x \in [a, b]_q \iff ((2^k + x - a \bmod q) \geq 2^k) \wedge ((x - b - 1 \bmod q) \geq 2^k) \tag{12}$$

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

$$((2^k + x - a \bmod q) \geq 2^k) \iff x \in [a, a + \Delta]_q$$
$$((x - b - 1 \bmod q) \geq 2^k) \iff x \in [b - \Delta, b]_q$$

Since by assumption $b - a \bmod^{\pm} q \leq \Delta$, 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 (12).

From (12), 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+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:

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

Algorithm 32 RangeTestShares

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

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

1: $(A_1, \ldots, A_n) \leftarrow (x_1 + 2^k - a \bmod q, x_2, \ldots, x_n)$
2: $(B_1, \ldots, B_n) \leftarrow (x_1 - b - 1 \bmod q, x_2, \ldots, x_n)$
3: $(y_1, \ldots, y_n) \leftarrow \mathsf{ArithmeticToBoolean}(q, (A_1, \ldots, A_n))$
4: $(z_1, \ldots, z_n) \leftarrow \mathsf{ArithmeticToBoolean}(q, (B_1, \ldots, B_n))$
5: $(u_1, \ldots, u_n) \leftarrow \mathsf{SecAnd}(1, (\mathsf{MSB}(y_1), \ldots, \mathsf{MSB}(y_n)), (\mathsf{MSB}(z_1), \ldots, \mathsf{MSB}(z_n)))$
6: **return** $(u_1, \ldots, u_n)$