High-order Table-based Conversion Algorithms 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 Masking is the main countermeasure against side-channel attacks on embedded devices. For cryptographic algorithms that combine Boolean and arithmetic masking, one must therefore convert between the two types of masking, without leaking additional information to the attacker. In this paper we describe a new high-order conversion algorithm between Boolean and arithmetic masking, based on table recomputation, and provably secure in the ISW probing model. We show that our technique is particularly efficient for masking structured LWE encryption schemes such as Kyber and Saber. In particular, for Kyber IND-CPA decryption, we obtain an order of magnitude improvement compared to existing techniques.

1 Introduction

Implementation of post-quantum cryptography. Most public-key cryptography deployed today is based on RSA and ECC and will not withstand attacks by a quantum computer. Consequently the National Institute of Standards and Technology (NIST) initiated in 2017 a public effort to standardize post-quantum cryptography. The process has now entered its third round, during which the implementation results of the remaining candidates must be evaluated. For embedded devices, one must therefore devise countermeasures against side-channel attacks. First introduced by Kocher [Koc96], a side-channel attack consists in recovering the secret-key by analyzing the running time, power consumption or electromagnetic emanation, during several executions of a cryptographic algorithm. Several works have demonstrated the effectiveness of side-channel attacks against post-quantum cryptography [TE15], for example against the BLISS signature scheme [BHLY16,EFGT17] and lattice-based encryption [PPM17,HCY20,XPRO20]. Consequently the NIST requested the scientific community to assist in the evaluation of the final round-3 submissions against side-channel attacks, as an important criterion for standardization [MAA+20]. In this paper we consider side-channel countermeasures for public-key encryption schemes based on the ring-learning with errors (ring-LWE) problem, such as two of the finalists Kyber [BDK+18,ABD+21] and Saber [BMD+21].

The masking countermeasure. Masking is an effective countermeasure against side-channel attacks. For security against first-order attacks, every variable x in the circuit can be masked by a random r into $x' = x \oplus r$. In that case the two shares x' and r are manipulated separately, which prevents first-order attacks since all intermediate variables considered separately have a uniform distribution [CJRR99]. However it is still possible to perform a second-order attack combining information leakage about the two shares x' and r; this usually requires a much larger amount of side-channel traces, see for example [OMHT06].

To prevent such higher-order attacks, a generalization of the masking countermeasure consists in splitting any variable x into n shares with $x = x_1 \oplus \cdots \oplus x_n$. The shares x_i must then be processed separately in order to avoid any leakage of information about the original variable x. To formally argue about the security provided by the masking countermeasure, Ishai, Sahai and Wagner introduced in [ISW03] the probing model, by considering an adversary who can probe at most t wires in a circuit. Using the masking countermeasure with $n = 2t + 1$ shares, they showed how to transform any boolean circuit C into a circuit of size $\mathcal{O}(|C| \cdot t^2)$ that is perfectly secure against such adversary. Moreover, in [DDF14], it was shown that security in the probing model implies security against noisy leakage, under the assumption that every variable leaks independently.

Security in the probing model is usually proven by simulation: one must show that any set of t probes can be perfectly simulated without knowing the original variables of C. To facilitate the writing of security proofs, Barthe *et al.* introduced the notions of (Strong) Non-Interference (NI/SNI), to allow easy composition of gadgets [BBD+16]. The authors proved the t-SNI property for the original ISW multiplication gadget. They also showed that with some additional mask refreshing, a circuit C can be made secure against t probes with $n = t+1$ shares only, instead of $n = 2t + 1$ shares in [ISW03].

While block-ciphers such as AES are typically protected using Boolean masking as above [RP10], lattice-based schemes often require a combination of arithmetic and Boolean masking. This implies that conversions between arithmetic and Boolean masking play an essential role in masked implementations of lattice-based schemes. Such conversions were previously used in block-ciphers and hash functions combining Boolean and arithmetic operations (such as SHA-1). However, they were based solely on power-of-two moduli, while many lattice-based schemes use a prime modulus (as in Kyber). The conversions must therefore be adapted in the context of post-quantum cryptography.

First-order conversion algorithms. The first conversion algorithms were proposed by Goubin in [Gou01], with security against first-order attacks. The Boolean to arithmetic conversion is efficient and has an optimal complexity $\mathcal{O}(1)$. The conversion from arithmetic to Boolean masking is less efficient as its complexity is $\mathcal{O}(k)$ for conversion from arithmetic masking modulo 2^k. This was later improved to $\mathcal{O}(\log k)$ in [CGTV15]; however in practice for $k = 32$ the number of operations was similar.

A table-based conversion from arithmetic to Boolean masking was described in [CT03], for first-order security only. For a small value of k, the conversion can be done by a simple table look-up, using a pre-computed table. This is similar to the classical first-order randomized SBox table countermeasure [CJRR99]. More precisely, the algorithm uses a randomized pre-computed table $T : \mathbb{Z}_{2^k} \to \{0,1\}^k$ which is initialized as follows. First, one generates a random mask $r \leftarrow \mathbb{Z}_{2^k}$. As a second step, one computes $T[v] = (v + r) \oplus r$ for all $v \in \mathbb{Z}_{2^k}$. Then, given an arithmetically masked value $A = x - r \pmod{2^k}$, one obtains a Boolean masking x' of x by simply reading the table T at index A, *i.e.* $x' = T[A]$; this gives $x' = (A + r) \oplus r = x \oplus r$ as required. The same randomized table can be used multiple times; therefore once the table has been initialized for all possible values in $\mathbb{Z}_{2^k}$, each conversion is a simple table look-up.

The authors also showed how to extend the technique to convert variables of $k = \delta \cdot \ell$ bits, by propagating the carry by blocks of ℓ bits. However there was a flaw in their algorithm: they computed the carry table modulo 2^ℓ only, instead of modulo $2^{k-\ell}$; therefore the algorithm is incorrect for $\delta > 2$; this mistake was identified and corrected by Debraize in [Deb12]. In [Deb12],

the author described multiple first-order conversion algorithms, but one of them was recently found insecure in [BDV21], who described two corrected algorithms.

High-order conversion algorithms. The first conversion algorithms secure against high-order attacks were described in [CGV14], with complexity $\mathcal{O}(n^2 \cdot k)$ for n shares and k-bit variables. The authors described conversions algorithms in both directions, with a security proof in the ISW probing model. Using the technique from [CGTV15], the complexity can be improved to $\mathcal{O}(n^2 \cdot \log k)$ as in the first-order case, however in practice the number of operations for $k = 32$ is also similar. The technique described in [CGV14,CGTV15] considers arithmetic masking modulo 2^k. This was later extended in [BBE+18] to arithmetic masking modulo any integer q, in the context of masking the GLP lattice-based signature scheme; the complexity is also $\mathcal{O}(n^2 \cdot k)$ or $\mathcal{O}(n^2 \cdot \log k)$ for a k-bit integer q. More precisely, the authors provided the extension of [CGV14,CGTV15] to arbitrary modulus with cubic complexity in n; the extension with quadratic complexity in n is provided in [SPOG19].

The approach used in [CGV14] to perform the Boolean to arithmetic conversion requires to first perform an arithmetic to Boolean conversion. An alternative, more direct approach is described in [SPOG19], also with complexity $\mathcal{O}(n^2 \cdot k)$. It also works with arithmetic masking modulo an arbitrary q. The technique is based on a 1-bit Boolean to arithmetic masking conversion with complexity $\mathcal{O}(n^2)$. Such 1-bit Boolean to arithmetic conversion is interesting in the context of lattice-based cryptography, for masking the re-encryption of the message, and for masking the binomial sampling.

Finally, a high-order Boolean to arithmetic conversion algorithm was described in [Cor17], and later simplified in [BCZ18], with complexity $\mathcal{O}(2^n)$; that is, the complexity is independent from the size of the k-bit variable that must be converted. The technique can be seen as a high-order extension of the original first-order Boolean to arithmetic algorithm from [Gou01]. Although the complexity is exponential in n, for small values of n the algorithm is at least one order of magnitude faster than [CGV14,CGTV15]. However for algorithms in [Cor17,BCZ18], the arithmetic masking is modulo 2^k only; we do not know how to extend the technique to any modulus q. We summarize in Table 1 the complexities of the conversions in both directions, for first-order attacks, and high-order attacks.

The randomized table countermeasure. For protecting the computation of an SBox against side-channel attacks, the classical first-order randomized table countermeasure from [CJRR99] was extended to high-order in [Cor14]. The high-order table recomputation countermeasure works as follows. Given a k-bit SBox S, one generates a table T with 2^k rows, where each row consists of n shares. Given as input n shares x_i such that $x = x_1 \oplus \cdots \oplus x_n$, the goal is to compute an n-sharing of $y = S(x)$, without leaking information about x. The rows of T are initialized with $T(u) = (S(u), 0, \ldots, 0)$ for all $u \in \{0,1\}^k$, and one maintains the invariant such that after Step i the rows of T are n-encodings of the rows of S, but shifted by $x_1 \oplus \cdots \oplus x_i$. For this one incrementally shifts the rows of the table by the successive input shares $x_1, \ldots, x_{n-1}$; the n-encodings on each row are refreshed between every shift. In the end, the rows of the table have been shifted by $x_1 \oplus \cdots \oplus x_{n-1}$, so it suffices to read the table T at the row $u = x_n$ to get the n output shares y_i corresponding to $y = S(x)$.

The countermeasure was proven secure against t probes in the ISW model with $n = 2t + 1$ shares, and later proven SNI in [CRZ18], so that it can work with $n = t + 1$ shares only. The authors also described a variant where the number of output shares in the randomized table T

is progressively increased from 1 to n, which saves a factor 2 in running time. For a k-bit SBOX and n shares, the countermeasure has complexity $\mathcal{O}(2^k \cdot n^2)$.

	Direction	Mod q	First-order complexity	High-order complexity	Memory complexity
[Gou01]	B $\to$ A	✗	$\mathcal{O}(1)$	-	$\mathcal{O}(1)$
	A $\to$ B	✗	$\mathcal{O}(k)$	-	$\mathcal{O}(1)$
[Cor17,BCZ18]	B $\to$ A	✗	-	$\mathcal{O}(2^n)$	$\mathcal{O}(2^n)$
[CGV14,CGTV15, BBE+18,SPOG19]	B $\to$ A, A $\to$ B	✓	-	$\mathcal{O}(n^2 \cdot k)$ $\mathcal{O}(n^2 \cdot \log k)$	$\mathcal{O}(n)$
[SPOG19]	(1-bit) B $\to$ A	✓	-	$\mathcal{O}(n^2)$	$\mathcal{O}(n)$
Algorithm 5	B $\to$ A	✓	-	$\mathcal{O}(n^2 \cdot k)$	$\mathcal{O}(n)$
Algorithm 3	(1-bit) B $\to$ A	✓	-	$\mathcal{O}(n^2)$	$\mathcal{O}(n)$
Algorithm 10	A $\to$ B	✗	-	$\mathcal{O}(n^2 \cdot k)$	$\mathcal{O}(n)$
Algorithm 11	A $\to$ (1-bit) B	✓	-	$\mathcal{O}(n^2)$	$\mathcal{O}(n)$
Algorithm 12	A $\to$ (1-bit) B	✓	-	$\mathcal{O}(n^2 \cdot \log n)$	$\mathcal{O}(n)$

Table 1. Complexities of Boolean vs arithmetic conversions in both directions, for first-order attacks, and for high-order attacks with n shares. For all algorithms the complexity is for k-bit register, except for Algorithm 11 which uses 2^k-bit registers. We indicate the Mod q property when the arithmetic masking can be modulo any q, not only modulo 2^k.

First contribution: high-order table-based conversion. Our first contribution is to extend the table-based conversion algorithm between Boolean and arithmetic masking of [CT03] from first-order to any order. For this we extend the high-order table recomputation countermeasure from [Cor14] recalled above. Namely we observe that in [Cor14], the incremental shifting of the rows of the table T can be performed according to any additive group G, not only for the xor operation in $\{0,1\}^k$. For example we can work modulo 2^k as input, which automatically gives a high-order conversion from arithmetic to Boolean masking. Similarly, the n-encoding of the rows of T as output can be according to any group law, not only for the xor in $\{0,1\}^k$. This implies that we can easily convert from Boolean to arithmetic masking modulo any integer q, which is useful in the context of lattice-based cryptography (see below).

More generally, our extended table recomputation countermeasure allows computing any function $f : G \to H$, for any group G as input and any group H as output. Given as input an n-sharing of $x = x_1 + \cdots + x_n \in G$, we can compute n outputs shares $y_i \in H$ such that $y_1 + \cdots + y_n = f(x_1 + \cdots + x_n)$, while being secure in the ISW probing model against $t = n - 1$ probes. By selecting the right groups G and H, we can therefore obtain high-order secure conversion algorithms between Boolean and arithmetic masking. To convert from Boolean to arithmetic masking modulo 2^k, we take $G = \{0,1\}^k$ and $H = \mathbb{Z}_{2^k}$ and we obtain $y_1 + \cdots + y_n = x_1 \oplus \cdots \oplus x_n \pmod{2^k}$ as required. Similarly for arithmetic to Boolean conversion we take $G = \mathbb{Z}_{2^k}$ and $H = \{0,1\}^k$, and we obtain $y_1 \oplus \cdots \oplus y_n = x_1 + \cdots + x_n \pmod{2^k}$ as required.

The main advantage of the table-based approach for conversions is its flexibility, as we can choose any groups G and H and any function $f : G \to H$. However the running time

complexity is $\mathcal{O}(n^2 \cdot |G|)$. This implies that for k-bit Boolean or arithmetic masking, the generic complexity is $\mathcal{O}(n^2 \cdot 2^k)$. Fortunately for specific groups G as input one can do much better. When converting from Boolean masking with $G = \{0,1\}^k$, we describe a simple optimization with complexity $\mathcal{O}(n^2 \cdot k)$ only, as in [CGV14]. For the other direction, we first describe a technique to compute a shift by ℓ bits on some arithmetically masked value, with complexity $\mathcal{O}(n^2 \cdot \ell)$. This is independently useful for the masking of Saber, which requires the computation of logical shifts. Our technique is based on an extension of the carry propagation technique from [CT03] to high-order security. From the arithmetic shift, it is then easy to obtain a conversion from arithmetic masking modulo 2^k to Boolean masking, again with complexity $\mathcal{O}(n^2 \cdot k)$ as in [CGV14]. See Table 1 for a summary of our conversion algorithms.

Moreover we describe two optimizations for the specific case of 1-bit Boolean masking, which is useful in the context of lattice-based cryptography. For converting from 1-bit Boolean masking to arithmetic masking modulo any q, our complexity becomes $\mathcal{O}(n^2)$ only, as in [SPOG19], instead of $\mathcal{O}(n^2 \cdot \log q)$ with [BBE+18]. In the other direction, we show how to efficiently compute a threshold function th from arithmetic masking modulo 2^k, whose result is a 1-bit Boolean masking. This corresponds to the decryption function in lattice-based cryptosystems. In that case, our optimization consists in putting each column of the table in a single register; the resulting complexity is $\mathcal{O}(n^2)$ only, assuming that we have access to 2^k-bit registers (see Algorithm 11 in Table 1). In practice, for both optimizations we obtain at least an order of magnitude improvement compared to the techniques in [CGV14,BBE+18].

Second contribution: high-order masking of lattice-based encryption schemes. Our second contribution is to apply our table-based conversion algorithms to the masking of lattice-based encryption schemes such as Kyber and Saber. Recall that for the IND-CCA decryption, one must perform the following operations according to the Fujisaki-Okamoto (FO) transform:

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 m
3. Polynomial comparison between c and c'.

We first consider the IND-CPA decryption of the ciphertext c (Step 1). For ring-LWE encryption the ciphertext $c = (c_1, c_2)$ is decrypted with the private-key s using $m = \mathsf{th}(c_1 - s \cdot c_2)$, where th is the threshold function $\mathsf{th} : \mathbb{Z}_q \to \{0,1\}$ where $\mathsf{th}(x) = 1$ if $x \in (q/4, 3q/4)$ and $\mathsf{th}(x) = 0$ otherwise. The threshold function is actually applied independently on each coefficient of the polynomial $u = c_1 - s \cdot c_2$ modulo q. When the private-key s is arithmetically masked modulo q with n shares, we obtain n shares for $u = u_1 + \cdots + u_n \pmod q$, and we must therefore convert from an arithmetically masked u modulo q into a 1-bit Boolean masked $m = m_1 \oplus \cdots \oplus m_n = \mathsf{th}(u)$. For this one could use our generic table-based approach with the function $f = \mathsf{th}$ and $f : \mathbb{Z}_q \to \{0,1\}$. However the complexity would be $\mathcal{O}(n^2 \cdot q)$, which is prohibitive for large q. Therefore we describe an optimization in which we first perform a modulus switching from masking modulo q to masking modulo 2^k (for a small k), while maintaining a negligible probability of decryption error.[1] We can then convert from arithmetic masking modulo 2^k into 1-bit Boolean masking, which recovers the Boolean masked message m. This optimization has complexity $\mathcal{O}(n^2 \cdot \log n)$ (see Algorithm 12 in Table 1), instead of $\mathcal{O}(n^2 \log q)$ with [BBE+18]. In practice we obtain an order of magnitude improvement in the IND-CPA decryption of Kyber.

[1] this is required to achieve CCA-security [DNR04].

We also consider the masking of re-encryption of m into a ciphertext c', and the masking of the binomial sampling (Step 2). To encrypt a Boolean masked message $m \in \{0,1\}$, we can use our generic table-based Boolean to arithmetic modulo q conversion algorithm. In that case the complexity is $\mathcal{O}(n^2)$ as in [SPOG19]. The same holds for the masking of the binomial sampling, which is easily computed as the sum of independent 1-bit Boolean to arithmetic modulo q conversions, as in [SPOG19]. In practice we obtain a similar level of efficiency as in [SPOG19], and an order of magnitude improvement compared to [BBE$^+$18].

In summary, our table-based approach for conversion between Boolean and arithmetic masking provides significant efficiency improvement in the context of lattice-based cryptography, especially for IND-CPA decryption (Step 1), while being relatively easy to implement. We provide a detailed comparison with existing conversion algorithms. We leave the high-order masking of the polynomial comparison (Step 3) for future work.

Prime vs power-of-two modulus. The main difference between Kyber and Saber is that Kyber uses a prime modulus while Saber uses a power-of-two modulus. In practice, the implementations in [MGTF19] and [GR19] observed a significant performance overhead for prime moduli compared to power-of-two moduli with regards to masking. Our work shows that the difference is not so significant. Firstly, when converting into an arithmetic masking, with our table-based technique the value of the arithmetic modulus q is irrelevant. Secondly, when converting from an arithmetic masking, in case of decryption we can easily perform a modulus switching to a power-of-two modulus, at the cost of a negligible increase in the decryption error. Such modulus switching has complexity $\mathcal{O}(n)$ for n shares, so for large n its complexity is negligible compared to the rest of the algorithm. In sections 8.5 and 10.4, we provide a detailed comparison of the computational cost of masking for prime moduli vs power-of-two moduli, in the context of ring-LWE encryption.

Practical implementation. We have benchmarked a plain C implementation of our techniques, and the conversion algorithms in [CGV14,BBE$^+$18] and [SPOG19] for comparison. We describe the results in Section 10. The code is publicly available at

`https://github.com/fragerar/HOTableConv`

1.1 Related work on masking ring-LWE encryption scheme

First-order masking. In [RRVV15], the authors described a first-order masking of IND-CPA decryption with $n = 2$ shares. They describe a relatively complex masked decoder to compute $m_1 \oplus m_2 = \mathsf{th}(u_1 + u_2)$. The decoder only works for half of the inputs, so it must be restarted up to 16 times with a certain shift $\delta \in [0, q-1]$.

In [OSPG18], the authors describe a first-order masking of the full IND-CCA decryption. The IND-CPA decryption part is based on first converting the arithmetic masking modulo q into an arithmetic sharing modulo 2^k, and then converting from arithmetic to Boolean masking. The re-encryption and binomial sampling are also masked to first-order. The polynomial comparison between x and $y = y_1 + y_2 \pmod q$ is done by checking that $H(x - y_1) = H(y_2)$. While efficient, the techniques seem relatively difficult to generalize to high-order.

In [BDK$^+$20], the authors describe a first-order masked implementation of Saber, with only a 2.5x overhead factor. They introduce an optimized arithmetic to arithmetic conversion algorithm (A2A) for performing logical shifts. Their algorithm is based on securely propagating the

carry with a pre-computed table, by adapting the techniques from [CT03,Deb12]. Their A2A conversion only applies to first-order masking. In this paper, we describe in Section 5 a generalization to high-order masking of the A2A computation of the logical shift, based on table recomputation.

In [FBR+21], the authors describe a first-order masking of Kyber and Saber, for both software and hardware implementations. For the masking of the Compress function in Kyber, they describe a technique based on modulus switching. They show that the error induced by the modulus switching can be eliminated by using more precision and then truncating, using an arithmetic to Boolean conversion. We use a similar modulus switching technique in Section 8.2, but extended to high-order masking.

High-order masking. In [SPOG19], the authors describe a very interesting technique to convert from 1-bit Boolean masking to arithmetic masking modulo q, with complexity $\mathcal{O}(n^2)$, for security against $t = n - 1$ probes. As an application, they obtain a high-order k-bit Boolean to arithmetic conversion algorithm with complexity $\mathcal{O}(k \cdot n^2)$, and also a high-order masked binomial sampling algorithm with complexity $\mathcal{O}(k \cdot n^2)$, where k is the length of the bit-vectors. The 1-bit Boolean to arithmetic masking modulo q is based on the following equation for $x, y \in \{0, 1\}$:

$$x \oplus y = x + y - 2 \cdot x \cdot y$$

which was already considered in [OSPG18] for first-order conversion. From the above equation, if we already have an arithmetic sharing of both x and y, we can obtain an arithmetic sharing of $x \oplus y$. Such arithmetic sharing can be modulo any integer q, as long as it encodes an element in $\{0, 1\}$. Namely the product $x \cdot y$ can be computed with n arithmetic shares modulo q, as in the And gate in [ISW03], with complexity $\mathcal{O}(n^2)$. Using a recursive approach similar to [CGV14], one obtains a 1-bit Boolean to arithmetic masking modulo q conversion, with complexity $\mathcal{O}(n^2)$. The authors of [SPOG19] actually describe an iterative approach, still with complexity $\mathcal{O}(n^2)$.

In [BDH+21], the authors describe an attack against the first-order masked ciphertext comparison in [OSPG18]. The attack is based on the fact that the ciphertext comparison in [OSPG18] is performed iteratively on different parts of the ciphertext, and the output of the first comparison leaks sensitive information to the attacker. The attack does not apply against the ciphertext comparison used for the protection of Saber in [BDK+20], which implements only a single check. The authors of [BDH+21] also describe a similar attack against the high-order polynomial comparison from [BPO+20], which proceeds in sets of ℓ coefficients and the pass/fail bit is unmasked for every set. They also describe a clever variant attack that does not use any side-channel information. To prevent these attacks, the polynomial comparison should be an atomic operation that does not reveal partial comparison results on a subset of the coefficients.

In [BGR+21], the authors describe the first completely masked implementation of Kyber, secure against first-order and higher-order attacks. For the IND-CPA decryption, the authors consider the $\mathsf{Compress}^s_q(x)$ function that outputs 0 if $x < q/2$ and 1 otherwise; this is a shifted function of the $\mathsf{Compress}_q(x, 1)$ from Kyber. They show that $\mathsf{Compress}^s_q(x) = x_{11} \oplus (\neg x_{11} \cdot x_{10} \cdot x_9 \cdot (x_8 \oplus (\neg x_8 \cdot x_7)))$, where x_i is the i-th bit of x. Therefore they proceed by first converting from arithmetic masking modulo q to Boolean masking, using [BBE+18]. Then the above function can be computed with high-order secure implementations of the And and Xor gadgets. In Appendix D.2 we describe a slightly simpler approach, still based on the arithmetic modulo q to Boolean masking conversion from [BBE+18]. The authors also describe a high-order secure polynomial comparison algorithm, which compares uncompressed masked polynomials

with compressed public polynomials, so that the ciphertext compression from Kyber does not need to be explicitly masked.

2 Security definitions

We recall below the t-NI and t-SNI security notions introduced in [BBD+16]. We consider a gadget taking as input a single n-tuple $(x_i)_{1\leq i\leq n}$ of shares, and outputting a single n-tuple $(y_i)_{1\leq i\leq n}$. Given a subset $I \subset [1, n]$, we denote by $x_{|I}$ all elements x_i such that $i \in I$.

Definition 1 (t-NI security [BBD+16]). *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 [BBD+16]). *Let G be a gadget taking as input the n shares $(x_i)_{1\leq i\leq n}$, and outputting n shares $(z_i)_{1\leq i\leq n}$. The gadget G is said to be t-SNI secure if for any set of t_1 probed intermediate variables and any subset $\mathcal{O}$ of output indices, such that $t_1 + |\mathcal{O}| \leq t$, there exists a subset I of input indices that satisfies $|I| \leq t_1$, such that the t_1 intermediate variables and the output variables $z_{|\mathcal{O}}$ can be perfectly simulated from $x_{|I}$.*

The main benefit of the t-SNI security definition is that it allows easy composition of gadgets [BBD+16]. By proving the t-SNI property of individual gadgets, we obtain that the full circuit is secure against t probes, using $n = t+1$ shares. In this paper we prove the t-NI or t-SNI property of all our gadgets. Note that a t-NI gadget is easily converted into t-SNI by applying a t-SNI mask refreshing as output (see [BBD+16]), making it suitable for composition with other gadgets in a larger circuit. For our generic high-order table-based conversion algorithm (Section 3), the t-SNI security proof is essentially the same as in [CRZ18]. For our more specialized gadgets, the t-NI or t-SNI properties follow almost directly from the t-SNI of our generic conversion algorithm.

3 Generic high-order table-based conversion algorithm

In this section we introduce our generic high-order table-based conversion algorithm, as an extension of the table recomputation countermeasure from [Cor14]. We consider two additive groups G and H and a function $f : G \to H$. Our algorithm takes as input n shares $x_1, \ldots, x_n \in G$ and outputs n shares $y_1, \ldots, y_n \in H$ such that:

$$y_1 + \cdots + y_n = f(x_1 + \cdots + x_n)$$

We stress that the function f does not need to have any special property, except being efficiently computable. In particular it need not be a group homomorphism, as in general the groups G and H will not be homomorphic. Note that the high-order SBox computation algorithm from [Cor14] is a particular case with $G = H = \{0, 1\}^k$ and $f(x) = S(x)$.

The algorithm consists in progressively shifting a randomized table T, using the input shares $x_1, \ldots, x_{n-1}$ for the successive shifts. The randomized table T has $|G|$ rows, and each row is a vector of n shares, which encodes over H the function $f(x)$, but progressively shifted by $x_1, \ldots, x_{n-1} \in G$. Eventually one reads the table at index x_n, which gives an n-sharing (y_i) over H of $f(x_1 + \cdots + x_n)$ as required. Between every shift, the n shares of every row are refreshed using the same mask refreshing as in [RP10], but over the group H.

As we will see in more details in Section 4, for a Boolean to arithmetic conversion algorithm, one will take $G = \{0,1\}^k$ and $H = \mathbb{Z}_{2^k}$. Then by identifying k-bit strings and integers modulo 2^k and taking f the identity function, we obtain $y_1 + \cdots + y_n \bmod 2^k = x_1 \oplus \cdots \oplus x_n$ as required. Similarly, for an arithmetic to Boolean conversion, one takes $G = \mathbb{Z}_{2^k}$ and $H = \{0,1\}^k$ and obtains $y_1 \oplus \cdots \oplus y_n = x_1 + \cdots + x_n \bmod 2^k$ as required; see Section 6 for more details.

Algorithm 1 $\mathsf{Convert}_{G,H,f}$

Input: $x_1, \ldots, x_n \in G$
Output: $y_1, \ldots, y_n \in H$ such that $y_1 + \cdots + y_n = f(x_1 + \cdots + x_n)$

1: **for all** $u \in G$ **do** $T(u) \leftarrow \big(f(u), 0, \ldots, 0\big) \in H^n$
2: **for** $i = 1$ to $n-1$ **do**
3: **for all** $u \in G$ **do** $T'(u) \leftarrow T(u + x_i)$
4: **for all** $u \in G$ **do** $T(u) \leftarrow \mathsf{Refresh}_H\big(T'(u)\big)$
5: **end for**
6: $(y_1, \ldots, y_n) \leftarrow \mathsf{Refresh}_H\big(T(x_n)\big)$
7: **return** $y_1, \ldots, y_n$

Algorithm 2 $\mathsf{Refresh}_H$

Input: $x_1, \ldots, x_n \in H$
Output: $y_1, \ldots, y_n \in H$ such that $y_1 + \cdots + y_n = x_1 + \cdots + x_n$

1: $y_n \leftarrow x_n$
2: **for** $j = 1$ to $n-1$ **do**
3: $r_j \leftarrow H$
4: $y_j \leftarrow x_j + r_j$
5: $y_n \leftarrow y_n - r_j$
6: **end for**
7: **return** $y_1, \ldots, y_n$

We provide a pseudocode description in Algorithm 1 above. The algorithm uses two temporary tables T and T' in RAM, with $|G|$ rows, where each row contains a vector of n elements in H. The table T is initialized at Line 1 with $T(u) \leftarrow \big(f(u), 0, \ldots, 0\big) \in H^n$. Given an encoding $\mathbf{v} = (v_1, \ldots, v_n)$ with n shares in H, we denote by

$$\sum(\mathbf{v}) = v_1 + \cdots + v_n$$

the encoded element in H. This implies that initially we have $\sum(T(u)) = f(u)$ for all rows $u \in G$. For the first index $i = 1$, the table is shifted at Line 3 by x_1 into T', which gives $\sum(T'(u)) = f(u + x_1)$ for all $u \in G$. Note that the shift is performed according to the group law in G. The rows are then refreshed at Line 4 using Algorithm 2, and we still have $\sum(T(u)) = f(u + x_1)$. More generally, after the shift by $x_1, \ldots, x_i$ we obtain at Line 4:

$$\sum(T(u)) = f(u + x_1 + \cdots + x_i)$$

for all $u \in G$, and after all the input shares $x_1, \ldots, x_{n-1}$ have been processed we have:

$$\sum(T(u)) = f(u + x_1 + \cdots + x_{n-1})$$

Therefore from the final look-up table $(y_1, \ldots, y_n) \leftarrow \mathsf{Refresh}_H\big(T(x_n)\big)$, we obtain that $\sum(\mathbf{y}) = \sum(T(x_n)) = f(x_n + x_1 + \cdots + x_{n-1})$. This gives $y_1 + \cdots + y_n = f(x_1 + \cdots + x_n)$, which proves the correctness of the algorithm.

The $\mathsf{Refresh}_H$ is the same as in [RP10] and [Cor14], except that we work in the group H instead of $\{0,1\}^k$. Note that the $\mathsf{Refresh}_H$ algorithm is not SNI; the only required property is that any subset of $n-1$ shares is uniformly and independently distributed (see Lemma 2 in Appendix A).

Complexity. We assume that a group operation in G and H takes unit time, as well as randomness generation and table transfer. For n shares, the number of operations of $\mathsf{Refresh}_H$ is $3n-3$. The time complexity of Algorithm 1 is therefore:

$$\begin{aligned} N_{\mathsf{convert}} &= |G| \cdot (n + (n-1) \cdot (1 + n + 3n - 3)) + 3n - 3 \\ &= |G| \cdot (4n^2 - 5n + 2) + 3n - 3 \simeq 4 \cdot |G| \cdot n^2 \end{aligned}$$

The asymptotic complexity is therefore $\mathcal{O}(|G| \cdot n^2)$. The memory complexity is $\mathcal{O}(|G| \cdot n)$. The algorithm requires $(n-1) \cdot (|G| \cdot (n-1) + 1)$ randoms.

Security. We prove that our algorithm achieves the t-SNI definition (Definition 2). One can therefore use our algorithm inside a more complex construction and achieve security against t probes with $n = t+1$ shares. The proof is essentially the same as in [CRZ18] and is provided in Appendix A.

Theorem 1 ($(n-1)$-SNI of $\mathsf{Convert}_{G,H,f}$). *For any subset $O \subset [1,n]$ and any t_1 intermediate variables with $|O| + t_1 < n$, the output variables $y_{|O}$ and the t_1 intermediate variables can be perfectly simulated from the input variables $x_{|I}$, with $|I| \leq t_1$.*

4 Table-based high-order Boolean to arithmetic conversion

In this section we consider the case of Boolean to arithmetic conversion. We first describe the straightforward application of the generic table-based conversion algorithm from G to H from Section 3. However the main drawback is that its complexity is $\mathcal{O}(|G| \cdot n^2)$, where $|G|$ is the group order. With k-bit Boolean masking as input we have $G = \{0,1\}^k$, and the complexity is therefore $\mathcal{O}(2^k \cdot n^2)$. Fortunately for specific groups G this complexity can be reduced to $\mathcal{O}(k \cdot n^2)$. In this section we consider the easiest case with the conversion from Boolean to arithmetic masking. We consider the other direction in Section 6.

4.1 Direct approach

We consider the straightforward application of Algorithm 1 to high-order Boolean to arithmetic conversion, which can be used for small values of k. We consider an integer q. We identify k-bit strings with integers in the interval $[0, 2^k[$. Algorithm 3 below describes a Boolean to arithmetic masking conversion algorithm such that given $x_1, \ldots, x_n \in \{0,1\}^k$ as input, we obtain $y_1, \ldots, y_n \in \mathbb{Z}_q$ as output, with, $x_1 \oplus \cdots \oplus x_n = y_1 + \cdots + y_n \pmod q$. The $(n-1)$-SNI security follows directly from Theorem 1.

Algorithm 3 BooleanToArithmetic

Input: $k \in \mathbb{Z}$ and $x_1, \ldots, x_n \in \{0,1\}^k$
Output: $y_1, \ldots, y_n \in \mathbb{Z}_q$ such that $y_1 + \cdots + y_n \bmod q = x_1 \oplus \cdots \oplus x_n$

1: **for all** $u \in \{0,1\}^k$ **do** $T(u) \leftarrow \big(u \bmod q, 0, \ldots, 0\big)$
2: **for** $i = 1$ to $n-1$ **do**
3: **for all** $u \in \{0,1\}^k$ **do** $T'(u) \leftarrow T(u \oplus x_i)$
4: **for all** $u \in \{0,1\}^k$ **do** $T(u) \leftarrow \mathsf{Refresh}_{\mathbb{Z}_q}\big(T'(u)\big)$
5: **end for**
6: $(y_1, \ldots, y_n) \leftarrow \mathsf{Refresh}_{\mathbb{Z}_q}\big(T(x_n)\big)$
7: **return** $y_1, \ldots, y_n$

Algorithm 4 $\mathsf{Refresh}_{\mathbb{Z}_q}$

Input: $x_1, \ldots, x_n \in \mathbb{Z}_q$
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: $r_j \leftarrow \mathbb{Z}_q$
4: $y_j \leftarrow x_j + r_j \bmod q$
5: $y_n \leftarrow y_n - r_j \bmod q$
6: **end for**
7: **return** $y_1, \ldots, y_n$

Complexity. We do not take into account the reductions modulo q. The operation count is the same as for Algorithm 1, with $|G| = 2^k$, which gives:

$$N_{\mathsf{BA}}(k,n) = 2^k \cdot (4n^2 - 5n + 2) + 3n - 3 \simeq 2^{k+2} \cdot n^2$$

The memory complexity is $\mathcal{O}(2^k \cdot n)$. The algorithm requires $(n-1) \cdot (2^k \cdot (n-1) + 1)$ randoms in $\mathbb{Z}_q$.

4.2 Optimization of high-order Boolean to arithmetic conversion

The main drawback of the previous generic algorithm is that its complexity is $\mathcal{O}(2^k \cdot n^2)$, which is prohibitive for large k, for example $k = 32$ in HMAC-SHA1. In this section we describe a simple optimization with complexity $\mathcal{O}(k \cdot n^2)$. It consists in converting each bit of the k-bit input separately and adding the result.

Assume that we must convert a Boolean masking $x = x_1 \oplus \cdots \oplus x_n \in \{0,1\}^k$ to arithmetic masking $x = y_1 + \cdots + y_n \pmod q$. We write the binary decomposition of each x_i as

$$x_i = \bigoplus_{j=0}^{k-1} 2^j \cdot x_i^{(j)}$$

where $x_i^{(j)}$ is the j-th bit of x_i. We have:

$$x = \bigoplus_{i=1}^{n} x_i = \bigoplus_{i=1}^{n} \bigoplus_{j=0}^{k-1} 2^j \cdot x_i^{(j)} = \bigoplus_{j=0}^{k-1} 2^j \cdot \bigoplus_{i=1}^{n} x_i^{(j)} = \sum_{j=0}^{k-1} 2^j \cdot \bigoplus_{i=1}^{n} x_i^{(j)}$$

Note that $x^{(j)} := \bigoplus_{i=1}^{n} x_i^{(j)}$ is the j-th bit of x. We now perform an independent table-based Boolean to arithmetic conversion for each of the k variables $x^{(j)}$. More precisely, applying Algorithm 3 on the Boolean shares $x_i^{(j)}$, we obtain n arithmetic shares $y_i^{(j)}$ for each $0 \leq j < k$:

$$x^{(j)} = \bigoplus_{i=1}^{n} x_i^{(j)} = \sum_{i=1}^{n} y_i^{(j)} \pmod{q}$$

This gives:

$$x = \sum_{j=0}^{k-1} 2^j \cdot \bigoplus_{i=1}^{n} x_i^{(j)} = \sum_{j=0}^{k-1} 2^j \cdot \sum_{i=1}^{n} y_i^{(j)} = \sum_{i=1}^{n} \sum_{j=0}^{k-1} 2^j \cdot y_i^{(j)} \pmod{q}$$

and therefore letting $y_i := \sum_{j=0}^{k-1} 2^j \cdot y_i^{(j)}$ for all $1 \leq i \leq n$, we obtain $x = y_1 + \cdots + y_n \pmod{q}$ as required. The algorithm is formally described in Algorithm 5 below.

Algorithm 5 Optimized BooleanToArithmetic (BAopti)

Input: $x_1, \ldots, x_n \in \{0,1\}^k$
Output: $y_1, \ldots, y_n \in \mathbb{Z}_q$ such that $x_1 \oplus \cdots \oplus x_n = y_1 + \cdots + y_n \bmod q$

1: **for** $i = 1$ to n **do** $y_i \leftarrow 0$
2: **for** $j = 0$ to $k - 1$ **do**
3: **for** $i = 1$ to n **do** $z_i \leftarrow (x_i \gg j)\ \&\ 1$
4: $(y_1^{(j)}, \ldots, y_n^{(j)}) \leftarrow \mathsf{BooleanToArithmetic}(1, z_1, \ldots, z_n)$
5: **for** $i = 1$ to n **do** $y_i \leftarrow y_i + (y_i^{(j)} \ll j) \bmod q$
6: **end for**
7: **return** $y_1, \ldots, y_n$

Complexity. Algorithm 5 computes the sum of k applications of Algorithm 3 with 1-bit input. More generally, one can group the conversions by ℓ bits. The number of operations is then:

$$N_{\mathsf{BAopti}}(k, n) = n + \lceil k/\ell \rceil \cdot (4 \cdot n + N_{\mathsf{BA}}(\ell, n))$$

One can see that it is a bit more advantageous to group by $\ell = 2$ bits. In that case, we have $N_{\mathsf{BAopti}}(k, n) \simeq 8k \cdot n^2$. The memory complexity is $\mathcal{O}(n)$. Namely Algorithm 5 uses a table with only 2 rows ($\ell = 1$) or 4 rows ($\ell = 2$) of n-shared encodings, therefore our table-based approach has a small memory consumption. The algorithm requires $\simeq 2k \cdot n^2$ randoms.

Theorem 2 ($(n-1)$-SNI of BAopti). *For any subset $O \subset [1, n]$ and any t_1 intermediate variables with $|O| + t_1 < n$, the output variables $y_{|O}$ and the t_1 intermediate variables can be perfectly simulated from the input variables $x_{|I}$, with $|I| \leq t_1$.*

Proof. The $(n-1)$-SNI property follows from the $(n-1)$-SNI of each of the k independent table-based conversions (Theorem 1). Namely the corresponding output shares $y_i^{(j)}$ are combined independently for each share index $1 \leq i \leq n$. Therefore we can use the same output subset O for each intermediate output shares $(y_i^{(j)})_{1 \leq i \leq n}$ for $0 \leq j < k$. □

4.3 Comparison with existing techniques

***k*-bit Boolean to arithmetic modulo 2^k conversion.** The k-bit Boolean to arithmetic modulo 2^k conversion is the classical case. We use $k = 32$, as for example in HMAC-SHA1. As shown in Table 2, our operation count is comparable to [CGV14] and [SPOG19]. For small orders t, it is, as for [CGV14] and [SPOG19], much less efficient than [BCZ18]. We refer to appendices C for the operation count of [Gou01,BCZ18,CGV14,SPOG19].

B → A mod 2^{32}	Security order t								
	1	2	3	4	5	6	8	10	12
Goubin [Gou01]	7								
[BCZ18]		49	123	277	591	1 225	5 053	20 401	81 829
[CGV14] $32 \to 32$		2 119	3 847	6 827	10 265	14 124	24 230	37 402	52 809
[SPOG19] $32 \to 32$		1 434	2 520	3 894	5 556	7 506	12 270	18 186	25 254
Algorithm 5, $32 \to 32$		1 763	3 348	5 445	8 054	11 175	18 953	28 779	40 653

Table 2. Operation count for k-bit Boolean to arithmetic modulo 2^k conversion algorithms, up to security order $t = 12$, with $n = t + 1$ shares, for $k = 32$.

1-bit Boolean to arithmetic modulo 2^k conversion. The 1-bit Boolean to arithmetic conversion is useful in the context of ring-LWE encryption. Here we use $k = 13$, since this corresponds to the binomial sampling for Saber, which can be written as a sum modulo 2^k of 1-bit Boolean to arithmetic modulo 2^k conversions, as in [SPOG19]. We see in Table 3 that our operation count is comparable to [SPOG19], both methods having complexity $\mathcal{O}(n^2)$. Our operation count is an order of magnitude faster than [CGV14], which has complexity $\mathcal{O}(k \cdot n^2)$. Namely the approach in [CGV14] requires to perform an arithmetic to Boolean conversion first, which has complexity $\mathcal{O}(k \cdot n^2)$, so one cannot really take advantage of the 1-bit Boolean masking as input.

B → A mod 2^{13}	Security order t								
	1	2	3	4	5	6	8	10	12
Goubin [Gou01]	7								
[BCZ18]		49	123	277	591	1 225	5 053	20 401	81 829
[CGV14] $1 \to 13$		884	1 605	2 837	4 261	5 859	10 037	15 476	21 839
[SPOG19] $1 \to 13$		39	71	112	162	221	366	547	764
Algorithm 3, $1 \to 13$		52	101	166	247	344	586	892	1 262

Table 3. Operation count for 1-bit Boolean to arithmetic modulo 2^k conversion algorithms, up to security order $t = 12$, with $n = t + 1$ shares, for $k = 13$.

1-bit Boolean to arithmetic modulo q conversion. We use $q = 3329$, as this corresponds to the encryption of Kyber, and to the binomial sampling of Kyber. For [BBE+18], we must use a word size k such that $2q < 2^k$, so we take $k = 13$. As previously, our complexity is comparable to [SPOG19], and more than an order of magnitude faster than [BBE+18].

B → A mod q	Security order t								
	1	2	3	4	5	6	8	10	12
[BBE+18] $1 \to \text{mod } q$	755	2 111	3 875	6 522	9 577	13 235	22 445	34 152	48 076
[SPOG19] $1 \to \text{mod } q$	16	39	71	112	162	221	366	547	764
Algorithm 3, $1 \to \text{mod } q$	19	52	101	166	247	344	586	892	1 262

Table 4. Operation count for 1-bit Boolean to arithmetic modulo q conversion algorithms, up to security order $t = 12$, with $n = t + 1$ shares, for prime $q = 3329$.

5 Table-based shift of arithmetic masking

In this section we consider the table-based computation of a right shift over arithmetic shares. This can be used directly in Saber, and this will be used as a subroutine for the arithmetic to Boolean masking conversion (Section 6).

We consider a parameter $1 \leq \ell < k$. We consider the function $f(x)$ performing a right shift of a k-bit integer x by ℓ bits, more precisely $f : \mathbb{Z}_{2^k} \to \mathbb{Z}_{2^{k-\ell}}$ with:

$$f(x) = \left\lfloor \frac{x}{2^\ell} \right\rfloor \bmod 2^{k-\ell}$$

Our goal is to compute this right shift with arithmetic shares. More precisely, given as input $z = z_1 + \cdots + z_n \pmod{2^k}$, we want to obtain arithmetic shares $a_1, \ldots, a_n \in \mathbb{Z}_{2^{k-\ell}}$ such that

$$a_1 + \cdots + a_n = \left\lfloor \frac{z_1 + \cdots + z_n}{2^\ell} \right\rfloor \pmod{2^{k-\ell}}$$

This right shift will be used for our table-based conversion from arithmetic to Boolean masking with complexity $\mathcal{O}(k \cdot n^2)$. Namely we will perform a sequence of k/ℓ right shifts by ℓ bits, each time converting a block of ℓ bits from arithmetic to Boolean masking. The goal of the right shift is to propagate the carry from one block to the next; this is a natural generalization of the carry propagation technique used in [CT03] for first-order table-based conversion.

If we are only interested in doing a right shift by ℓ bits (as in Saber), a basic approach using [CGV14] consists in first performing an arithmetic to Boolean conversion, then in doing an easy logical right shift by ℓ bits of the Boolean shares, and eventually in converting back the result to arithmetic modulo $2^{k-\ell}$. The complexity is then $\mathcal{O}(k \cdot n^2)$, and therefore independent from ℓ.

In the following, we describe a table-based approach with complexity $\mathcal{O}(\ell \cdot n^2)$. Therefore we expect the approach to be more efficient than [CGV14] for small values of ℓ. We actually describe a first technique with complexity $\mathcal{O}(2^\ell \cdot n^3)$, and a second technique with complexity $\mathcal{O}(2^{2\ell} \cdot n^2)$. To obtain a linear complexity in ℓ, in both cases we can perform a sequence of ℓ shifts by $\ell' = 1$ bit each. The first technique has then complexity $\mathcal{O}(\ell \cdot 2^{\ell'} \cdot n^3) = \mathcal{O}(\ell \cdot n^3)$, while the second technique has complexity $\mathcal{O}(\ell \cdot 2^{2\ell'} \cdot n^2) = \mathcal{O}(\ell \cdot n^2)$. Because of a smaller constant in the $\mathcal{O}$, we expect the first technique to be more efficient for small n.

5.1 First approach with complexity $\mathcal{O}(2^\ell \cdot n^3)$

We consider the function $f : \mathbb{Z}_{2^k} \to \mathbb{Z}_{2^{k-\ell}}$ defined previously with $f(x) = \lfloor x/2^\ell \rfloor \bmod 2^{k-\ell}$, which corresponds to the $k - \ell$ most significant bits of x. We consider an arithmetic masking

$z = z_1 + \cdots + z_n \pmod{2^k}$ as input. Our goal is to obtain an arithmetic sharing of $f(z)$ modulo $2^{k-\ell}$. For all $1 \leq i \leq n$, we write $z_i = y_i \cdot 2^\ell + x_i$ with $0 \leq x_i < 2^\ell$. This gives:

$$z = 2^\ell \cdot \sum_{i=1}^{n} y_i + \sum_{i=1}^{n} x_i \pmod{2^k}$$

and therefore:

$$f(z) = \sum_{i=1}^{n} y_i + f\left(\sum_{i=1}^{n} x_i \bmod 2^k\right) \pmod{2^{k-\ell}} \tag{1}$$

The previous equation shows that to compute $f(z)$ (which corresponds to the $k-\ell$ most significant bits of z), we must compute the carry resulting from the addition of the ℓ-bit shares x_i, *i.e.* $f\left(\sum_{i=1}^{n} x_i \bmod 2^k\right)$. For this we apply our generic Algorithm 1 with inputs $x_1, \cdots, x_n$, $G = \mathbb{Z}_{2^k}$, $H = \mathbb{Z}_{2^{k-\ell}}$ and f, and we obtain an arithmetic masking of the resulting carry:

$$c_1 + \cdots + c_n = f(x_1 + \cdots + x_n \bmod 2^k) \pmod{2^{k-\ell}} \tag{2}$$

Combining (1) and (2), this gives:

$$f(z) = \sum_{i=1}^{n} (y_i + c_i) \pmod{2^{k-\ell}}$$

Therefore we have obtained an arithmetic sharing of the $k-\ell$ high-order bits of z.

For a naive implementation of Algorithm 1, the complexity of this step would be $\mathcal{O}(|G| \cdot n^2) = \mathcal{O}(2^k \cdot n^2)$. Therefore there would be no advantage compared to a generic table-based computation of the function f. However we note that since $0 \leq x_i < 2^\ell$ for all i, we actually have $0 \leq \sum_{i=1}^{n} x_i \leq n \cdot (2^\ell - 1)$ in (2). Therefore, when applying Algorithm 1, we do not need to store and randomize a full table with 2^k rows, as we can work with a much smaller table with $B = n \cdot (2^\ell - 1) + 1$ rows only. Thanks to this optimization the complexity becomes $\mathcal{O}(B \cdot n^2) = \mathcal{O}(2^\ell \cdot n^3)$. Moreover the table does not have to be cyclically shifted, only translated by x_i for each $1 \leq i \leq n-1$; this implies that a single table in memory is sufficient. Our method is described below in Algorithm 6.

Algorithm 6 Shift1

Input: $k \in \mathbb{N}^+, 1 \leq \ell < k$ and $z_1, \ldots, z_n \in \mathbb{Z}_{2^k}$
Output: $a_1, \ldots, a_n \in \mathbb{Z}_{2^{k-\ell}}$ such that $a_1 + \cdots + a_n = f(z_1 + \cdots + z_n) \pmod{2^{k-\ell}}$

1: **for** $i = 1$ to n **do** $x_i \leftarrow z_i \bmod 2^\ell$
2: **for** $u = 0$ to $n \cdot (2^\ell - 1)$ **do** $T(u) \leftarrow (u \gg \ell, 0, \ldots, 0)$
3: **for** $i = 1$ to $n - 1$ **do**
4: **for** $u = 0$ to $(n - i) \cdot (2^\ell - 1)$ **do**
5: $T(u) \leftarrow T(u + x_i)$
6: $T(u) \leftarrow \mathsf{Refresh}_{\mathbb{Z}_{2^{k-\ell}}}(T(u))$
7: **end for**
8: **end for**
9: $(c_1, \ldots, c_n) \leftarrow \mathsf{Refresh}_{\mathbb{Z}_{2^{k-\ell}}}(T(x_n))$
10: **for** $i = 1$ to n **do** $a_i \leftarrow (z_i \gg \ell) + c_i \bmod 2^{k-\ell}$
11: **return** $a_1, \ldots, a_n$

Complexity. The operation count is as follows:

$$\begin{aligned} N_{\mathsf{s1}}(\ell, n) &= n + n \cdot (2^\ell - 1) + 1 + \sum_{i=1}^{n-1} ((2^\ell - 1)(n-i) + 1) \cdot (n + 1 + 3(n-1)) \\ &\quad + 3(n-1) + 2n \\ &= 5n - 2 + n \cdot 2^\ell + (4n-2) \cdot \Big((2^\ell - 1) \cdot (n-1) \cdot n/2 + n - 1 \Big) \\ &= 2^\ell \cdot n \cdot (2n^2 - 3n + 2) - 2n^3 + 7n^2 - 2n \end{aligned}$$

The algorithm therefore requires $2^{\ell+1} \cdot n^3$ operations, neglecting low-order terms. By performing a sequence of ℓ shifts of 1-bit each, the number of operations is $\ell \cdot N_{\mathsf{s1}}(1, n) \simeq 2\ell \cdot n^3$, neglecting low-order terms. The memory complexity is $\mathcal{O}(n^2)$, since the table has $\mathcal{O}(n)$ rows. More precisely, with $\ell = 1$, the table has n rows of n-shared encodings, which corresponds to n^2 values in memory. In the next section we describe an alternative technique with memory complexity $\mathcal{O}(n)$ only. The number of randoms is $(n-1) \cdot n \cdot (n+1)/2$.

Security. The algorithm only achieves the $(n-1)$-NI property. To achieve the stronger $(n-1)$-SNI property, one can apply a $(n-1)$-SNI mask refreshing algorithm as output (see [BBD+16]). Such mask refreshing has complexity $\mathcal{O}(n^2)$ only, so this does not change the asymptotic complexity $\mathcal{O}(\ell \cdot n^3)$.

Theorem 3 ($(n-1)$-NI of Shift1). *Any set of $t_1 \leq n-1$ intermediate variables can be perfectly simulated from the input variables $z_{|I}$, with $|I| \leq t_1$.*

Proof. The table-based conversion algorithm up to Line 9 is the same as the $(n-1)$-SNI Algorithm 1, except that we are only performing a fraction of the computation. This implies that the adversary can only probe a subset of the variables, and therefore the algorithm remains $(n-1)$-SNI, and therefore $(n-1)$-NI. The global algorithm combines at Line 10 those output shares with a right shift of the input shares. Therefore the algorithm is $(n-1)$-NI. □

5.2 Second approach with complexity $\mathcal{O}(2^{2\ell} \cdot n^2)$

We consider again the function $f : \mathbb{Z}_{2^k} \to \mathbb{Z}_{2^{k-\ell}}$ defined previously with $f(x) = \lfloor x/2^\ell \rfloor \bmod 2^{k-\ell}$, for some parameter $1 \leq \ell < k$. We consider two integers $z_1, z_2 \in \mathbb{Z}_{2^k}$, and we write:

$$\begin{aligned} z_1 &= y_1 \cdot 2^\ell + x_1 \\ z_2 &= y_2 \cdot 2^\ell + x_2 \end{aligned}$$

where $0 \leq x_1, x_2 < 2^\ell$. To compute the carry in the sum of x_1 and x_2, we consider the function $g : (\mathbb{Z}_{2^\ell})^2 \to \mathbb{Z}_{2^{k-\ell}}$, with:

$$g(u, v) = \begin{cases} 0 & \text{if } u + v < 2^\ell \\ 1 & \text{if } u + v \geq 2^\ell \end{cases}$$

By propagating the carry from the sum of x_1 and x_2, we get:

$$f(z_1 + z_2) = y_1 + y_2 + g(x_1, x_2) \pmod{2^{k-\ell}} \tag{3}$$

We consider an arithmetic masking $z = z_1 + \cdots + z_n \pmod{2^k}$. For all $1 \leq i \leq n$, we write as previously $z_i = y_i \cdot 2^\ell + x_i$ with $0 \leq x_i < 2^\ell$. Equation (3) can be generalized to:

$$f(z_1 + \cdots + z_n) = \sum_{i=1}^{n} y_i + \sum_{j=1}^{n-1} g\left(\sum_{i=1}^{j} x_i \bmod 2^\ell,\ x_{j+1}\right) \pmod{2^{k-\ell}} \tag{4}$$

Now applying Algorithm 1 with $G = (\mathbb{Z}_{2^\ell})^2$, $H = \mathbb{Z}_{2^{k-\ell}}$ and g, we can obtain the following arithmetic masking for all $1 \leq j \leq n-1$:

$$c_{j,1} + \cdots + c_{j,n} = g\left(\sum_{i=1}^{j} x_i,\ x_{j+1}\right) \pmod{2^{k-\ell}}$$

Namely the input of g can be computed as a sum over the additive group $\mathbb{Z}_{2^\ell} \times \mathbb{Z}_{2^\ell}$:

$$c_{j,1} + \cdots + c_{j,n} = g\left(\sum_{i=1}^{j} (x_i, 0) + (0, x_{j+1})\right) \pmod{2^{k-\ell}} \tag{5}$$

Eventually by combining (4) and (5) we obtain:

$$f(z) = \sum_{i=1}^{n} \left(y_i + \sum_{j=1}^{n-1} c_{j,i}\right) \pmod{2^{k-\ell}}$$

Therefore as previously we have obtained an arithmetic sharing of the $k - \ell$ high-order bits of z, which means that we can perform a shift by ℓ bits over the arithmetic shares of z.

For each $1 \leq j < n$, Equation (5) can be evaluated using Algorithm 1 with complexity $\mathcal{O}(|G| \cdot j \cdot n) = \mathcal{O}(2^{2\ell} \cdot j \cdot n)$; namely there are $j+1$ input shares instead of n. Therefore the total complexity of the first step would be $\mathcal{O}(2^{2\ell} \cdot n^3)$, which is still cubic in n as previously. However it is possible to evaluate (5) in a more clever way. Namely we can keep the table randomization obtained up to $\sum_{i=1}^{j}(x_i, 0)$ when computing the new table randomization up to $\sum_{i=1}^{j+1}(x_i, 0)$. The complexity then becomes $\mathcal{O}(2^{2\ell} \cdot n^2)$. The algorithm is described formally below in Algorithm 7.

Algorithm 7 Shift2

Input: $k \in \mathbb{N}^+, 1 \leq \ell < k$ and $z_1, \ldots, z_n \in \mathbb{Z}_{2^k}$
Output: $a_1, \ldots, a_n \in \mathbb{Z}_{2^{k-\ell}}$ such that $a_1 + \cdots + a_n = f(z_1 + \cdots + z_n) \pmod{2^{k-\ell}}$

1: **for** $i = 1$ to n **do** $(a_i, x_i) \leftarrow (z_i \gg \ell, z_i \bmod 2^\ell)$
2: **for all** $(u, v) \in \mathbb{Z}_{2^\ell} \times \mathbb{Z}_{2^\ell}$ **do** $T((u, v)) \leftarrow \big((u + v) \gg \ell, 0, \ldots, 0\big) \in (\mathbb{Z}_{2^{k-\ell}})^n$
3: **for** $i = 1$ to $n - 1$ **do**
4: **for all** $(u, v) \in \mathbb{Z}_{2^\ell} \times \mathbb{Z}_{2^\ell}$ **do** $T'((u, v)) \leftarrow T((u + x_i, v))$
5: **for all** $(u, v) \in \mathbb{Z}_{2^\ell} \times \mathbb{Z}_{2^\ell}$ **do** $T((u, v)) \leftarrow \mathsf{Refresh}_{\mathbb{Z}_{2^{k-\ell}}}\big(T'((u, v))\big)$
6: $(c_1, \ldots, c_n) \leftarrow \mathsf{Refresh}_{\mathbb{Z}_{2^{k-\ell}}}\big(T((0, x_{i+1}))\big)$
7: **for** $j = 1$ to n **do** $a_j \leftarrow a_j + c_j \bmod 2^{k-\ell}$
8: **end for**
9: **return** $a_1, \ldots, a_n$

Complexity. The number of operations is given by:

$$\begin{aligned} N_{\mathsf{s2}}(\ell, n) &= 2n + 2^{2\ell+1} + (n-1) \cdot \left(2^{2\ell} \cdot (n+1+3(n-1)) + 3(n-1) + n\right) \\ &= 2n + 2^{2\ell}\left(2 + (n-1)(4n-2)\right) + (n-1)(4n-3) \\ &= 2^{2\ell}(4n^2 - 6n + 4) + 4n^2 - 5n + 3 \end{aligned}$$

The algorithm therefore requires $2^{2\ell+2} \cdot n^2$ operations, neglecting low-order terms. By performing a sequence of ℓ shifts of 1-bit each, the number of operations is therefore $\ell \cdot N_{\mathsf{s2}}(1, n) \simeq 20\ell \cdot n^2$, neglecting low-order terms. The memory complexity is $\mathcal{O}(n)$, instead of $\mathcal{O}(n^2)$ for Shift1. More precisely, with $\ell = 1$, the tables T and T' have 4 rows of n-shared encoding, so a total of $8n$ values. The number of randoms is $5(n-1)^2$.

Security. The Shift2 algorithm only achieves the $(n-1)$-NI property. To achieve the stronger $(n-1)$-SNI property, as previously one can apply a $(n-1)$-SNI mask refreshing algorithm as output, without changing the asymptotic complexity.

Theorem 4 ($(n-1)$-NI of Shift2). *Any set of $t_1 \leq n-1$ intermediate variables can be perfectly simulated from the input variables $z_{|I}$, with $|I| \leq t_1$.*

Proof. The security proof is very similar to the security proof of Theorem 1. It is easy to see that the computation of the shares c_i at Line 6 achieves the $(n-1)$-SNI property. Namely either a variable is probed between lines 4 and 5 and we include $i \in I$ to get the knowledge of x_i from z_i, or no variable is probed and we can perfectly simulate any proper subset of shares at Line 5, thanks to the mask refreshing. The same holds at Line 6 with the knowledge of x_{i+1}.

Therefore the computation of the shares c_i at Line 6 also achieves the weaker $(n-1)$-NI property, and as in the proof of Theorem 3, the combination of shares computed at Line 7 remains $(n-1)$-NI. □

5.3 Comparison with existing technique

For the shift by ℓ bits of an arithmetic masking modulo 2^k, we perform a concrete comparison between the $\mathcal{O}(k \cdot n^2)$ method using [CGV14] and our $\mathcal{O}(\ell \cdot n^2)$ method; see Table 5 below. As explained previously, when using [CGV14] (or [Gou01] at first-order), we first perform an arithmetic to Boolean conversion, then a right shift by ℓ bits of the Boolean shares, and eventually a Boolean to arithmetic conversion modulo $2^{k-\ell}$. We use $k = 13$ and $\ell = 3$ as in Saber. We see that our table-based algorithm is more efficient. In particular, Algorithm 6 with complexity $\mathcal{O}(\ell \cdot n^3)$ is more efficient for small orders (up to $t = 6$), while Algorithm 7 with complexity $\mathcal{O}(\ell \cdot n^2)$ is more efficient for high orders.

Shift	Security order t								
	1	2	3	4	5	6	8	10	12
Shift with [Gou01]	79								
Shift with [CGV14]		1 159	2 109	3 867	5 870	8 085	13 966	21 696	30 679
Shift1 (Algorithm 6)	72	207	456	855	1 440	2 247	4 671	8 415	13 767
Shift2 (Algorithm 7)	123	336	669	1 122	1 695	2 388	4 134	6 360	9 066

Table 5. Operation count for arithmetic shift with $k = 13$ bit input and a shift by $\ell = 3$ bits, computed as a sequence of ℓ shifts of 1 bit for our table-based countermeasure, with $n = t + 1$ shares.

6 Table-based high-order arithmetic to Boolean conversion

6.1 Direct approach for arithmetic modulo q

We consider the direct application of Algorithm 1 to high-order arithmetic to Boolean conversion. Given $x_1, \ldots, x_n \in \mathbb{Z}_q$ as input, we obtain $y_1, \ldots, y_n \in \{0,1\}^k$ as output, with $x_1 + \cdots + x_n \bmod q = y_1 \oplus \cdots \oplus y_n$. For this we have to assume that $q \leq 2^k$, since the sum $x_1 + \cdots + x_n \bmod q$ needs at least $\lceil \log_2 q \rceil$ bits for its representation. We provide the pseudocode description below.

Algorithm 8 ArithmeticToBoolean

Input: $q \in \mathbb{Z}$ and $x_1, \ldots, x_n \in \mathbb{Z}_q$
Output: $y_1, \ldots, y_n \in \{0,1\}^k$ such that $y_1 \oplus \cdots \oplus y_n = x_1 + \cdots + x_n \bmod q$

1: **for all** $u \in \mathbb{Z}_q$ **do** $T(u) \leftarrow (u, 0, \ldots, 0)$
2: **for** $i = 1$ to $n-1$ **do**
3: **for all** $u \in \mathbb{Z}_q$ **do** $T'(u) \leftarrow T(u + x_i \bmod q)$
4: **for all** $u \in \mathbb{Z}_q$ **do** $T(u) \leftarrow \mathsf{Refresh}_{\{0,1\}^k}(T'(u))$
5: **end for**
6: $(y_1, \ldots, y_n) \leftarrow \mathsf{Refresh}_{\{0,1\}^k}(T(x_n))$
7: **return** $y_1, \ldots, y_n$

Algorithm 9 $\mathsf{Refresh}_{\{0,1\}^k}$

Input: $x_1, \ldots, x_n \in \{0,1\}^k$
Output: $y_1, \ldots, y_n \in \{0,1\}^k$ such that $y_1 \oplus \cdots \oplus y_n = x_1 \oplus \cdots \oplus x_n$

1: $y_n \leftarrow x_n$
2: **for** $j = 1$ to $n-1$ **do**
3: $r_j \leftarrow \{0,1\}^k$
4: $y_j \leftarrow x_j \oplus r_j$
5: $y_n \leftarrow y_n \oplus r_j$
6: **end for**
7: **return** $y_1, \ldots, y_n$

The operation count is the same as for Algorithm 1, with $|G| = q$, which gives:

$$N_{\mathsf{AB}}(q, n) = q \cdot (4n^2 - 5n + 2) + 3n - 3 \simeq 4q \cdot n^2$$

The memory complexity is $\mathcal{O}(q \cdot n)$. The number of randoms is $(n-1) \cdot (q \cdot (n-1) + 1)$. The t-SNI security follows directly from Theorem 1.

6.2 Optimization for arithmetic modulo 2^k with secure shift

The main drawback of the previous generic algorithm is that its complexity is $\mathcal{O}(q \cdot n^2)$, which is prohibitive for large q. In this section we describe an optimization for $q = 2^k$ with complexity $\mathcal{O}(k \cdot n^2)$ instead of $\mathcal{O}(2^k \cdot n^2)$. The technique is based on the secure computation of the right shift from Section 5. We can use either Shift1 (Algorithm 6), with complexity $\mathcal{O}(k \cdot n^3)$, or Shift2 (Algorithm 7), with complexity $\mathcal{O}(k \cdot n^2)$.

Assume that we are given as input $z = z_1 + \cdots + z_n \pmod{2^k}$ and we must compute $s_1, \ldots, s_n \in \{0,1\}^k$ such that $s_1 \oplus \cdots \oplus s_n = z_1 + \cdots + z_n \pmod{2^k}$. For this we define a

parameter $1 \leq \ell < k$, and using one of the two Shift algorithms from Section 5, given as input the shares $z_1, \ldots, z_n \in \mathbb{Z}_{2^k}$, we obtain arithmetic shares $a_1, \ldots, a_n \in \mathbb{Z}_{2^{k-\ell}}$ such that:

$$\left\lfloor \frac{z}{2^\ell} \right\rfloor = \sum_{i=1}^{n} a_i \pmod{2^{k-\ell}}$$

By definition we have $z = \lfloor z/2^\ell \rfloor \cdot 2^\ell + (z \bmod 2^\ell)$. Therefore letting $x_i = z_i \bmod 2^\ell$ for all $1 \leq i \leq n$ we can write:

$$z = 2^\ell \cdot \sum_{i=1}^{n} a_i + \left(\sum_{i=1}^{n} x_i \bmod 2^\ell \right) \pmod{2^k} \tag{6}$$

Equation (6) shows that we have actually obtained two independent arithmetic sharing: an arithmetic sharing $(a_i)_{1 \leq i \leq n}$ of the $k-\ell$ high-order bits of z, and an arithmetic sharing $(x_i)_{1 \leq i \leq n}$ of the ℓ low-order bits of z. One can then directly convert the arithmetic sharing $(x_i)_{1 \leq i \leq n}$ into Boolean masking using Algorithm 8 from Section 6.1, with complexity $\mathcal{O}(2^\ell \cdot n^2)$. This gives a Boolean masking of the ℓ low-order bits of z. The process can be applied recursively with the $k - \ell$ high-order bits of z, starting now from the arithmetic sharing $(a_i)_{1 \leq i \leq n}$. Eventually one obtains a full Boolean masking of z. The algorithm is formally described in Algorithm 10 below.

Algorithm 10 Optimized ArithmeticToBoolean (ABopti)

Input: $k, \ell \in \mathbb{N}^+$ and $z_1, \ldots, z_n \in \mathbb{Z}_{2^k}$
Output: $s_1, \ldots, s_n \in \{0,1\}^k$ such that $s_1 \oplus \cdots \oplus s_n = z_1 + \cdots + z_n \bmod 2^k$

1: **if** $k \leq \ell$ **then**
2: $\quad (s_1, \ldots, s_n) \leftarrow \mathsf{ArithmeticToBoolean}(2^k, (z_1, \ldots, z_n))$
3: **else**
4: $\quad$ **for** $i = 1$ to n **do** $x_i \leftarrow z_i \bmod 2^\ell$
5: $\quad (a_1, \ldots, a_n) \leftarrow \mathsf{Shift}(k, \ell, (z_1, \ldots, z_n))$
6: $\quad (h_1, \ldots, h_n) \leftarrow \mathsf{ABopti}(k - \ell, \ell, (a_1, \ldots, a_n))$
7: $\quad (l_1, \ldots, l_n) \leftarrow \mathsf{ArithmeticToBoolean}(2^\ell, (x_1, \ldots, x_n))$
8: $\quad$ **for** $i = 1$ to n **do** $s_i \leftarrow ((h_i \ll \ell) + l_i) \bmod 2^k$
9: **end if**
10: **return** $s_1, \ldots, s_n$

Operation count. The number of operations is given by:

$$N_{\mathsf{ABo}} = \lceil k/\ell \rceil \cdot (n + N_{\mathsf{s}}(\ell, n) + N_{\mathsf{AB}}(2^\ell, n) + 2n)$$

where $N_{\mathsf{s}}(\ell, n)$ is the number of operations for the shift algorithm, using either Shift1 or Shift2. Taking $\ell = 1$, we obtain $N_{\mathsf{ABo}} \simeq 2k \cdot n^3$ using Shift1, neglecting low-order terms, and $N_{\mathsf{ABo}} \simeq 28k \cdot n^2$ using Shift2. In summary, the complexity of our arithmetic to Boolean conversion using Shift2 is $\mathcal{O}(k \cdot n^2)$. The memory complexity using Shift2 is $\mathcal{O}(n)$. In particular, the table-based arithmetic to Boolean conversion at Line 7 uses 2 rows of n encodings. The number of randoms with $\ell = 1$ is $k \cdot (n-1) \cdot (7n-6)$.

Theorem 5 ($(n-1)$-SNI of ABopti). *For any subset $O \subset [1, n]$ and any t_1 intermediate variables with $|O| + t_1 < n$, the output variables $s_{|O}$ and the t_1 intermediate variables can be perfectly simulated from the input variables $z_{|I}$, with $|I| \leq t_1$.*

Proof. The $(n-1)$-SNI property is proven recursively for the number of blocks $j = \lceil k/\ell \rceil$. For $j = 1$ this follows from the $(n-1)$-SNI of the generic Algorithm 8. Assuming the $(n-1)$-SNI property for j blocks, the conversion for $j+1$ blocks combines at Line 8 the output h_i of the $(n-1)$-SNI conversion with j blocks and the output l_i of the $(n-1)$-SNI conversion of Algorithm 8. Moreover the $(n-1)$-SNI conversion with j blocks uses as input the shares $(a_i)_{1 \le i \le n}$ from the $(n-1)$-NI algorithms Shift1 or Shift2, and therefore the composition starting from the input shares z_i remains $(n-1)$-SNI. Therefore the conversion with $j+1$ blocks is also $(n-1)$-SNI. This proves the property. □

6.3 Optimization with table in registers

We describe an optimization of Algorithm 8, where the j-th column of the table is stored in a single register R_j for $1 \le j \le n$. The cyclic shift of the rows of the table by input share x_i then corresponds to a simple rotation of each register R_j. In the following we consider the arithmetic to Boolean conversion with k bits as input and 1 bit as output, as will be used in Section 8.2 for the IND-CPA decryption of lattice-based encryption.

More precisely we consider the computation of a function $f : \mathbb{Z}_{2^k} \to \{0,1\}$. One can consider for example the threshold function $f(x) = \lfloor x/2^{k-1} \rfloor$. Given as input n arithmetic shares $x_1, \ldots, x_n \in \mathbb{Z}_{2^k}$, our goal is to compute 1-bit Boolean shares $y_1, \ldots, y_n \in \{0,1\}$ such that $y_1 \oplus \cdots \oplus y_n = f(x_1 + \cdots + x_n \bmod 2^k)$.

Since we must store every column of the table with 2^k rows in a single register, each register must have 2^k bits. We denote by $R_j[u]$ the u-th bit of register R_j, for $0 \le u < 2^k$ and $1 \le j \le n$. Then Line 1 of Algorithm 8 becomes $R_1[u] = f(u)$ for $0 \le u < 2^k$, and $R_j = 0$ for $2 \le j \le n$. The rotation of the table at Line 3 becomes a rotation of all registers R_j by x_i positions to the right. The refreshing of the rows of the table at Line 4 becomes a mask refreshing of the shares $(R_1, \ldots, R_n)$ with 2^k-bit randoms. Eventually we must read and refresh the row x_n of the table (Line 6 of Algorithm 8), so we simply read the x_n-th bit of each register R_j. We refer to Algorithm 11 for a formal description. We denote by $\mathsf{ROR}[a](R)$ the cyclic rotation of a 2^k-bit register R by a bits to the right.

Algorithm 11 ArithmeticToBoolean, register optimization (ABreg)

Input: $x_1, \ldots, x_n \in \mathbb{Z}_{2^k}$
Output: $y_1, \ldots, y_n \in \{0,1\}$ such that $y_1 \oplus \cdots \oplus y_n = f(x_1 + \cdots + x_n \bmod 2^k)$

1: **for all** $u \in \mathbb{Z}_{2^k}$ **do** $R_1[u] \leftarrow f(u)$
2: **for all** $2 \le j \le n$ **do** $R_j \leftarrow 0$.
3: **for** $i = 1$ to $n-1$ **do**
4: **for** $j = 1$ to n **do** $R_j \leftarrow \mathsf{ROR}[x_i](R_j)$
5: **for** $j = 1$ to $n-1$ **do**
6: $r \leftarrow \{0,1\}^{2^k}$, $R_j \leftarrow R_j \oplus r$, $R_n \leftarrow R_n \oplus r$
7: **end for**
8: **end for**
9: $(y_1, \ldots, y_n) \leftarrow \mathsf{Refresh}_{\{0,1\}}(R_1[x_n], \ldots, R_n[x_n])$
10: **return** $y_1, \ldots, y_n$

Operation count. We do not count the 2^k operations of Line 1, since the value eventually stored in the register R_1 can be pre-computed. The number of operations is given by:

$$\begin{aligned} N_{\mathsf{ABreg}}(n) &= (n-1)(n+3(n-1)) + n + 3(n-1) \\ &= 4n^2 - 3 \cdot n \simeq 4 \cdot n^2 \end{aligned}$$

Using 2^k-bit registers, the complexity of the countermeasure is therefore $\mathcal{O}(n^2)$, assuming that generating a 2^k-bit random takes unit time. The memory complexity is $n+1$ registers of 2^k bits.

Obviously this optimization can only work for small values of k. In the comparison with existing techniques (sections 6.4 and 8.5), we use the following more realistic estimate of operation count, assuming a 32-bit processor. We assume that a register operation (or random generation) takes 1 operation for 32-bit ($k = 5$), and more generally 2^{k-5} operations for 2^k bits, for $k \geq 5$. The time complexity then becomes $N'_{\mathsf{ABreg}}(n,k) = 2^{k-5} \cdot (4n^2 - 3n)$. The number of 32-bit randoms is $2^{k-5} \cdot (n-1)^2 + n - 1$ for $k \geq 5$.

For $k = 5$, the implementation only requires $n + 1$ registers of 32-bits. More generally, for $k \geq 5$, the memory complexity is $(n+1) \cdot 2^{k-5}$ registers of 32 bits.

Security. We prove below the $(n-1)$-SNI property of Algorithm 11. We stress that we do not put two shares from the same encoding into the same register. Otherwise the attacker could obtain information from multiple shares of the same encoding using a single probe on a given register, which would break the $(n-1)$-SNI property.

Theorem 6 ($(n-1)$-SNI of ABreg). *For any subset $O \subset [1,n]$ and any t_1 intermediate variables with $|O| + t_1 < n$, the output variables $y_{|O}$ and the t_1 intermediate variables can be perfectly simulated from the input variables $x_{|I}$, with $|I| \leq t_1$.*

Proof. The proof is essentially the same as the proof of Theorem 1. The only difference is that by probing register a R_j the adversary gets the full j-th column of the table, instead of a single cell only. Such probe is simulated in the same way by putting the index j in J for every such probe. □

Extensions. The technique is easily extended to arithmetic masking modulo any q as input, not only $q = 2^k$. In that case, one must perform two shifts for each register, instead of a single rotation for $q = 2^k$. Moreover, the technique is easily extended to k-bit Boolean masking as output, instead of 1-bit. In that case, one must use registers of size $k \cdot 2^k$ bits instead of 2^k.

6.4 Comparison with existing techniques

Arithmetic modulo 2^k to k-bit Boolean conversion. As in Section 4.3, we consider the classical case of arithmetic modulo 2^k to k-bit Boolean conversion, and we use $k = 32$ as in HMAC-SHA1. We see in Table 6 that in that case our table-based technique is less efficient than [CGV14].

A mod $2^{32} \to$ B	Security order t								
	1	2	3	4	5	6	8	10	12
Goubin [Gou01]	165								
[CGV14] $32 \to 32$		1 132	2 070	4 030	6 218	8 597	15 053	23 655	33 572
Alg. 10 with Shift1		3 872	8 528	15 840	26 384	40 736	83 168	147 744	239 072
Alg. 10 with Shift2		5 536	10 752	17 760	26 560	37 152	63 712	97 440	138 336

Table 6. Operation count for arithmetic modulo 2^k to k-bit Boolean conversion algorithms, up to security order $t = 12$, with $n = t + 1$ shares and $k = 32$. We use $\ell = 2$ for Shift1, and $\ell = 1$ for Shift2.

Arithmetic modulo 2^k to 1-bit Boolean conversion, for small k. As we will see in Section 8.2, arithmetic modulo 2^k to 1-bit Boolean conversion is interesting in the context of ring-LWE IND-CPA decryption, in order to compute the threshold function $\mathsf{th} : \mathbb{Z}_{2^k} \to \{0, 1\}$, with $\mathsf{th}(x) = 1$ if $x \in [2^{k-2}, 3 \cdot 2^{k-2})$ and $\mathsf{th}(x) = 0$ otherwise.

Such threshold function th can be computed directly using our Algorithm 11 from the previous section, since the algorithm works for any function f. Alternatively, to compute th with [CGV14], we write $\mathsf{th}(x) = \mathsf{th}'(x - 2^{k-2})$ where $\mathsf{th}'(x) = 1$ if $x \in [0, 2^{k-1})$ and 0 otherwise. Thus, $\mathsf{th}'(x)$ is the complement of the most significant bit of x. Therefore we first subtract 2^{k-2} to the first arithmetic share of x, and perform the arithmetic to Boolean conversion from [CGV14]. Finally we extract the most significant bit of each Boolean share, and complement the first share; see Appendix D.1 for more details. We see in Table 7 that for $k = 6$ (see Section 8 for a motivation of this choice of k), we obtain a significant improvement compared to [CGV14].

A mod $2^6 \to$ B	Security order t								
	1	2	3	4	5	6	8	10	12
Goubin [Gou01]	38								
[CGV14] $6 \to 1$		226	411	786	1 207	1 663	2 895	4 531	6 416
Algorithm 11	20	54	104	170	252	350	594	902	1 274

Table 7. Operation count for arithmetic modulo 2^k to 1-bit Boolean conversion algorithms, up to security order $t = 12$, with $n = t + 1$ shares and $k = 6$.

7 Ring-LWE encryption and the masking countermeasure

7.1 Ring-LWE encryption

We first recall the principle of ring-LWE encryption [LPR10]. We then consider Module-LWE (M-LWE) encryption and two finalists of the NIST PQC standardization: Kyber and Saber. The two independent candidates share a lot of similarities as they are both M-LWE-based encryption schemes.

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.

Ring-LWE IND-CPA encryption. Let $\mathcal{R}$ and $\mathcal{R}_q$ denote the rings $\mathbb{Z}[X]/(X^d+1)$ and $\mathbb{Z}_q[X]/(X^d+1)$ respectively, for some $d \in \mathbb{Z}$ 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

$$\begin{aligned} c_1 &= a \cdot e_1 + e_2 \\ c_2 &= t \cdot e_1 + e_3 + \lfloor q/2 \rceil \cdot m \end{aligned} \tag{7}$$

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:

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

The distribution χ can be based on binomial sampling, which is easier to implement than the discrete Gaussian distribution [ADPS16]. More precisely, one can compute each polynomial coefficient as the difference between the Hamming weights of two random κ-bit strings, for some parameter κ.

Module-LWE IND-CPA encryption. A public-key encryption scheme based on the module learning-with-errors problem (M-LWE) in module lattices [LS15] is parameterized by a ring $\mathcal{R}_q$, a module rank l and a distribution χ outputting "small" elements in $\mathcal{R}$. In Kyber and Saber, we use $\mathcal{R} = \mathbb{Z}[X]/(X^d+1)$ and $\mathcal{R}_q = \mathbb{Z}_q[X]/(X^d+1)$, and χ is a distribution outputting polynomials with coefficients independently drawn from a centered binomial distribution of fixed parameter. We denote vectors and matrices by boldfaced variables. Let $\mathbf{s}$ and $\mathbf{e}$ be elements of $\mathcal{R}^l$ sampled from χ^l and $\mathbf{A}$ a uniformly random element of $\mathcal{R}_q^{l \times l}$. The public key is $\mathbf{t} = \mathbf{A} \cdot \mathbf{s} + \mathbf{e} \in \mathcal{R}_q^l$ and the secret key is $\mathbf{s}$. To CPA-encrypt a message $m \in \mathcal{R}$ with binary coefficients, one computes $(\mathbf{c}_1, c_2) \in \mathcal{R}_q^l \times \mathcal{R}_q$ such that

$$\begin{aligned} \mathbf{c}_1 &= \mathbf{A}^T \cdot \mathbf{r} + \mathbf{e}_1 \\ c_2 &= \mathbf{t}^T \cdot \mathbf{r} + e_2 + \lfloor q/2 \rceil \cdot m \end{aligned}$$

where $\mathbf{r}$ and $\mathbf{e}_1$ are sampled from χ^d and e_2 from χ . To decrypt a ciphertext $(\mathbf{c}_1, c_2)$, one computes:

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

The last approximation holds because elements sampled from χ are small. To recover the original message m, as previously one applies coefficient-wise the threshold function th.

CCA-secure KEM. Both Kyber and Saber aim to actually construct a CCA-secure key encapsulation mechanism from their IND-CPA encryption. They both use a variant of the FO transform [FO99]. Since the details do not matter for masking, we describe a simplified version. To encrypt a random session key K, one first generates a random message m; this message is then encrypted with the IND-CPA encryption scheme using a random tape $r = H_1(m)$ derived from the message itself. The ciphertext is still $c = (\mathbf{c}_1, c_2)$, while the session-key is $K = H_2(m, c)$. To decrypt, one first recovers m from $(\mathbf{c}_1, c_2)$; one can then re-encrypt m using the same randomness $r = H_1(m)$ to get a new ciphertext c'; one then checks that $c = c'$; in that case one outputs $K = H_2(m, c)$, and $\perp$ otherwise.

Specificities of Kyber. Kyber instantiates the M-LWE-based encryption scheme described above with $d = 256$, a prime $q = 3329$ and a centered binomial distribution of parameters 2 or 3 [ABD+21]. The designers also introduced a compression function for reducing the size of the ciphertext. Indeed, since the decryption has to tolerate a certain amount of error to properly recover the message, a trade-off between correctness and ciphertext size can be made by purposefully dropping some low-order bits of $(\mathbf{c}_1, c_2)$. The compression function is defined as $\mathsf{Compress}_q(x, a) = \lceil (2^a/q) \cdot x \rfloor \bmod 2^a$ and inverted by $\mathsf{Decompress}_q(x, a) = \lceil (q/2^a) \cdot x \rfloor$. Those functions are defined on scalars, and extended to polynomials and vectors of polynomials by applying them separately on each coefficient. We also note that the compression function with $a = 1$ is used to recover the message at the end of the IND-CPA decryption. Eventually, since q is a prime such that $q-1$ is divisible by d, for efficiency reasons, the number theoretic transform (NTT) can be used for polynomial multiplication [PG13]. Thus, the public-key, private-key and some part of the ciphertext are transmitted in the NTT domain. The NTT transform being a linear operation, it is usually not relevant for the definition of the masking scheme, so for simplicity we ignore it for the rest of the paper.

Specificities of Saber. Saber is based on the hardness on the Module Learning With Rounding (M-LWR) problem [BMD+21]. In a nutshell, instead of explicitly adding error terms $(\mathbf{e}, \mathbf{e}_1, e_2)$ sampled from the distribution χ, errors are deterministically added by applying a rounding to the value. For example, if the public-key is of the form $\mathbf{t} = \mathbf{A} \cdot \mathbf{s} + \mathbf{e}$ in an LWE-based scheme, the LWR equivalent will use $\mathbf{t} = \lfloor \mathbf{A} \cdot \mathbf{s} \rceil_p$, with $\lfloor \cdot \rceil_p$ a rounding function mapping $\mathbb{Z}_q$ to $\mathbb{Z}_p$ with $p < q$. In the case of Saber, both p and q are powers of two and the rounding function is basically a shift extracting the $\log_2(p)$ most significant bits of its input. This modulus switch from a power of two modulus to another one is also used to compress the ciphertext, and eventually to decode the noisy message.

7.2 Masking lattice-based encryption scheme

Masking IND-CPA decryption. We consider the masking of ring-LWE decryption against side-channel attacks. The secret-key $s \in \mathcal{R}$ is initially masked with n shares using $s = s_1 + \cdots + s_n \pmod q$ where $s_i \in \mathcal{R}_q$ for all $1 \leq i \leq n$. Given as input a ciphertext (c_1, c_2), instead of computing $u = c_2 - s \cdot c_1$ and then $m = \mathsf{th}(u)$ coefficient-wise, in the first step we can write:

$$\begin{aligned} u &= c_2 - (s_1 + \cdots + s_n) \cdot c_1 \pmod q \\ &= u_1 + \cdots + u_n \pmod q \end{aligned}$$

where $u_1 = c_2 - s_1 \cdot c_1$ and $u_i = -s_i \cdot c_1$ for all $2 \leq i \leq n$. Therefore we have obtained an arithmetic sharing of u modulo q. In the second step, one must compute n Boolean shares m_i of the message $m = m_1 \oplus \cdots \oplus m_n$ such that

$$m_1 \oplus \cdots \oplus m_n = \mathsf{th}(u_1 + \cdots + u_n) \tag{8}$$

without leaking information about $u = c_2 - s \cdot c_1$ in the process. Otherwise knowing u the adversary could recover the secret-key as $s = (c_2 - u)/c_1 \bmod q$. Computing the threshold function th securely over the shares u_i as in (8) is non-trivial, because th is a non-linear function from $\mathbb{Z}_q$ to $\{0, 1\}$. Moreover, as observed in [OSPG18], one should not eventually recombine the shares m_i into m, since otherwise knowing m the adversary can launch a CCA attack and recover the private-key s.

Masking IND-CCA decryption. As recalled previously, the IND-CCA decryption of a ring-LWE scheme (such as Kyber and Saber) performs the following operations according to the Fujisaki-Okamoto (FO) transform:

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 errors from m
3. Polynomial comparison between c and c'

More precisely, under a simplified version of the FO transform and with ring-LWE encryption as in (7), in the second step the message m is re-encrypted using error polynomials $(e_1, e_2, e_3) = H_1(m)$ to get a new ciphertext c', and one then checks that $c = c'$ before outputting the key $K = H_2(m, c)$.

We have considered the masked implementation of Step 1 (IND-CPA decryption) in the previous paragraph. As observed previously, the message m recovered at Step 1 must be kept in shared form with the shares m_i obtained from (8), otherwise knowing $m = m_1 \oplus \cdots \oplus m_n$ the adversary could launch a CCA attack. Therefore at Step 2 the hash-function $H_1(m)$ should be masked, and the binomial sampling for generating $(e_1, e_2, e_3) = H_1(m)$ should also be masked, which gives a masked re-encrypted ciphertext c'. Eventually the polynomial comparison between c and c' should be masked, and if $c = c'$ one must return a masked session key $K = H_2(m, c)$.

Since the IND-CCA decryption algorithm combines arithmetically masked values (such as s, e_1, e_2, e_3 and c') and Boolean masked values (such as m and K), this requires conversions between Boolean and arithmetic masking. In this paper, for the masking of ring-LWE encryption, we only consider the high-order masking of IND-CPA decryption (Step 1), and the re-encryption of m with the binomial sampling (Step 2). Namely our goal is to show that our table recomputation technique for high-order conversions is particularly effective in the context of ring-LWE encryption. We leave for further work the masking of the polynomial comparison (Step 3), and the description of a fully masked IND-CCA ring-LWE scheme secure against high-order attacks.

Finally, for simplicity we will focus on the high-order masking of operations performed on single elements in $\mathbb{Z}_q$, as for example the computation of the threshold function $\mathsf{th} : \mathbb{Z}_q \to \{0, 1\}$. Namely when dealing with polynomials as in ring-LWE scheme, and additionally with vectors and matrices of polynomials as in M-LWE and M-LWR schemes (as with Kyber and Saber), these operations are performed coefficient-wise and component-wise. Therefore the corresponding algebraic structure is irrelevant for the description of the masking scheme.

8 Masking ring-LWE IND-CPA decryption

8.1 Overview

As explained in Section 7.2, for masking the ring-LWE IND-CPA decryption, we must compute a threshold function th over arithmetic shares modulo q, with 1-bit Boolean shares as output. Namely for Kyber we must compute the threshold function $\mathsf{th} : \mathbb{Z}_q \to \{0,1\}$ with

$$\mathsf{th}(x) = \begin{cases} 0 & \text{if } (x \bmod^{\pm} q) \in [-q/4, q/4[\\ 1 & \text{otherwise.} \end{cases} \tag{9}$$

More precisely, given as input arithmetic shares $x_1, \ldots, x_n \in \mathbb{Z}_q$, we must compute 1-bit Boolean shares $y_1, \ldots, y_n \in \{0,1\}$ such that

$$y_1 \oplus \cdots \oplus y_n = \mathsf{th}(x_1 + \cdots + x_n \bmod q)$$

The computation of the threshold function for Saber is similar (see Appendix D.1).

For computing the threshold function th, we could apply our generic conversion algorithm from Section 3 with $G = \mathbb{Z}_q$, $H = \{0,1\}$ and the function $f : \mathbb{Z}_q \to \{0,1\}$ with $f = \mathsf{th}$. In that case, the time complexity is $\mathcal{O}(q \cdot n^2)$, and the memory consumption is $\mathcal{O}(q \cdot n)$. Both can be prohibitive for large q, for example with $q = 3329$ in Kyber, or with $q = 2^{10}$ in Saber.

We describe in the following an optimized technique based on modulus switching to a smaller modulus 2^ℓ, which enables to apply the fast table-based variant from Section 6.3. For this we slightly modify the decryption algorithms of Kyber, with a negligible increase in the decryption failure probability. We explain in Section 8.4 why the security proof for Kyber remain perfectly valid. Namely the IND-CCA security proof only depends on the decryption failure probability, and not on the specific decryption algorithm used. In other words, one can use any decryption algorithm, as long as the decryption failure probability remains neligible. We maintain a total decryption failure probability $\delta \leq 2^{-128}$ to guarantee the same level of security as in the original scheme, against both classical attacks and quantum attacks.

8.2 Threshold arithmetic modulo q to 1-bit Boolean

Our goal is to compute the function $\mathsf{th} : \mathbb{Z}_q \to \{0,1\}$ given by (9) but this time we require a correct computation of $\mathsf{th}(x)$ only for a large subset of $\mathbb{Z}_q$, not necessarily for the full $\mathbb{Z}_q$. More precisely, we require correct computation only for values of x which are not too close to the thresholds $\pm q/4$, by a relative factor Δ. More precisely we require correct decryption for $x \in R_{q,\Delta}$ with:

$$R_{q,\Delta} = \left\{ x \in \mathbb{Z}_q,\ |x \bmod^{\pm} q| < q \cdot \left(\frac{1}{4} - \Delta\right) \text{ or } |x \bmod^{\pm} q| > q \cdot \left(\frac{1}{4} + \Delta\right) \right\} \tag{10}$$

This means that there will be a small subset of $\mathbb{Z}_q$ for which the computation of the function th can be incorrect. We will see that for lattice-based schemes such as Kyber and Saber, the probability that $x \notin R_{q,\Delta}$ is negligible for small enough Δ, and therefore the decryption error will remain negligible for these two schemes.

Our goal is therefore to compute output shares $b_1, \ldots, b_n \in \{0,1\}$, such that when given input shares $x_1, \ldots, x_n \in \mathbb{Z}_q$ such that $x = x_1 + \cdots + x_n \in R_{q,\Delta}$, we are guaranteed to obtain a correct result, that is:

$$b_1 \oplus \cdots \oplus b_n = \mathsf{th}(x_1 + \cdots + x_n)$$

For this our strategy is to first perform a modulus switching into an arithmetic masking modulo a smaller 2^ℓ, and then to perform the (easier) conversion from arithmetic masking modulo 2^ℓ to Boolean masking via a threshold function f over $\mathbb{Z}_{2^\ell}$. More precisely, we first perform a modulus switching of all input shares x_i, by computing $y_i = \lfloor x_i \cdot 2^\ell/q \rceil$ for all $1 \le i \le n$. Note that this modulus switching can be computed by writing

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

and therefore y_i is the quotient of the Euclidean division of $x_i \cdot 2^{\ell+1} + q$ by $2q$. We obtain:

$$\sum_{i=1}^{n} y_i = \sum_{i=1}^{n} \left\lfloor \frac{2^\ell \cdot x_i}{q} \right\rceil = \sum_{i=1}^{n} \frac{2^\ell \cdot x_i}{q} + \varepsilon_i$$

where $|\varepsilon_i| \le 1/2$ for all $1 \le i \le n$. This gives:

$$y = \sum_{i=1}^{n} y_i = \frac{2^\ell \cdot x}{q} + \varepsilon \pmod{2^\ell}, \quad \text{where } |\varepsilon| \le n/2 \tag{11}$$

Therefore we have obtained an arithmetic masking of $y \in \mathbb{Z}_{2^\ell}$ where $y = 2^\ell \cdot x/q + \varepsilon \pmod{2^\ell}$ and the error $\varepsilon \in \mathbb{R}$ is such that $|\varepsilon| \le n/2$. In the second step, we apply the generic conversion algorithm from Section 3 with $G = \mathbb{Z}_{2^\ell}$, $H = \{0,1\}$ and the function $f : \mathbb{Z}_{2^\ell} \to \{0,1\}$ where

$$f(y) = \begin{cases} 0 & \text{if } (y \bmod^{\pm} 2^\ell) \in (-2^{\ell-2}, 2^{\ell-2}) \\ 1 & \text{otherwise.} \end{cases}$$

Our algorithm is formally described in Algorithm 12 below.

Algorithm 12 Arithmetic modulo q to 1-bit Boolean conversion (ThresholdAtoB)

Input: $x_1, \ldots, x_n \in \mathbb{Z}_q$
Output: $b_1, \ldots, b_n \in \{0,1\}$ such that $b_1 \oplus \cdots \oplus b_n = \mathsf{th}(x)$ for $x = x_1 + \cdots + x_n \in R_{q,\Delta}$
1: **for** $i = 1$ to n **do** $y_i \leftarrow \lfloor x_i \cdot 2^\ell/q \rceil$
2: $(b_1, \ldots, b_n) \leftarrow \mathsf{Convert}_{\mathbb{Z}_{2^\ell},\{0,1\},\mathsf{f}}(y_1, \ldots, y_n)$
3: **return** $b_1, \ldots, b_n$

Correctness and complexity. The following lemma proves the correctness of Algorithm 12 when the threshold function $\mathsf{th}(x)$ is computed on $x \in R_{q,\Delta}$, under the condition $n \le 2^{\ell+1} \cdot \Delta$.

Lemma 1. *Assume that $n \le 2^{\ell+1} \cdot \Delta$. The output of Algorithm 12 is correct if $x_1 + \cdots + x_n \in R_{q,\Delta}$.*

Proof. Assume that $x \in R_{q,\Delta}$ and let represent x in $(-q/2, q/2)$. Assume that $|x| < q \cdot (1/4 - \Delta)$. This implies:

$$\left| \frac{2^\ell}{q} \cdot x \right| < 2^{\ell-2} - \Delta \cdot 2^\ell \le 2^{\ell-2} - \frac{n}{2}$$

From (11), this implies that $|y \bmod^{\pm} 2^\ell| < 2^{\ell-2}$ and therefore $f(y) = \mathsf{th}(x) = 0$ as required.

Similarly, if $|x| \ge q \cdot (1/4 + \Delta)$, then $|x \cdot 2^\ell/q| > 2^{\ell-2} + \Delta \cdot 2^\ell \ge 2^{\ell-2} + n/2$ and therefore $|y \bmod^{\pm} 2^\ell| > 2^{\ell-2}$, which implies $f(y) = \mathsf{th}(x) = 1$ as required. This proves the correctness of Algorithm 12. □

From Lemma 1, it suffices to select an intermediate modulus 2^ℓ with

$$\ell = \lceil \log_2(n/\Delta) \rceil - 1 \tag{12}$$

to ensure correct computation of $\mathsf{th}(x)$ for $x \in R_{q,\Delta}$. The complexity of the arithmetic to Boolean conversion at Line 2 is therefore $\mathcal{O}(2^\ell \cdot n^2) = \mathcal{O}(n^3)$ using the generic conversion (Algorithm 8). Using the optimized arithmetic to Boolean conversion (Algorithm 10), the complexity becomes $\mathcal{O}(\ell \cdot n^2) = \mathcal{O}(n^2 \cdot \log n)$. The memory complexity remains $\mathcal{O}(n)$. Finally, using the optimization with table in registers from Section 6.3, the complexity is $\mathcal{O}(n^2)$ only, assuming that operations on registers of size 2^ℓ take unit time.

Security. The previous algorithm achieves the $(n-1)$-SNI property, thanks to the $(n-1)$-SNI property of the Convert algorithm.

8.3 Application to ring-LWE IND-CPA decryption

In this section we show how to efficiently mask the IND-CPA decryption of ring-LWE schemes. We explain how to tune the value Δ used in the definition of $R_{q,\Delta}$ in (10) so that the decryption error remains negligible for Kyber.

As explained in Section 7.2, for masking the ring-LWE IND-CPA decryption, the secret-key $s \in \mathcal{R}$ is initially masked with n shares using $s = s_1 + \cdots + s_n \pmod q$ where $s_i \in \mathcal{R}_q$ for all $1 \le i \le n$. Given as input a ciphertext (c_1, c_2), instead of computing $u = c_2 - s \cdot c_1$ and then $m = \mathsf{th}(u)$ coefficient-wise, in the first step we compute $u_1 = c_2 - s_1 \cdot c_1$ and $u_i = -s_i \cdot c_1$ for all $2 \le i \le n$, which gives an arithmetic sharing of $u = u_1 + \cdots + u_n \in \mathcal{R}_q$.

Therefore, in the second step, by applying Algorithm 12 coefficient-wise on the polynomial shares $u_i \in \mathcal{R}_q$, we obtain n boolean shares m_i of the message $m = m_1 \oplus \cdots \oplus m_n$ such that $m_1 \oplus \cdots \oplus m_n = \mathsf{th}(u_1 + \cdots + u_n)$, as required. To ensure a negligible decryption error, we must therefore ensure that all coefficients of u belong to the set $R_{q,\Delta}$ considered in the previous section, except with negligible probability.

Application to Kyber. The authors of the Kyber submission provide a Python script computing a tight upper bound on the decryption error probability δ. Following [HHK17], we say that $\mathsf{PKE} = (\mathsf{KeyGen}, \mathsf{Enc}, \mathsf{Dec})$ is $(1-\delta)$ correct if $\mathbb{E}[\max_{m \in \mathcal{M}} \Pr[\mathsf{Dec}(sk, \mathsf{Enc}(pk, m)) = m]] \ge 1 - \delta$, where the probability is over the randomness of Enc, and the expectation is over $(pk, sk) \leftarrow \mathsf{KeyGen}()$. More precisely, for an encryption of 0, the authors compute an upper-bound on the probability that any coefficient of $u = c_2 - \mathbf{s}^T \cdot \mathbf{c}_1$ is greater than $q/4$ in absolute value. From the definition of the set $R_{q,\Delta}$ in (10), it suffices to rerun the script with the bound $q \cdot (1/4 - \Delta)$ instead, in order to obtain the new decryption failure probability. We refer to Appendix B for more details.

For our implementations, we choose to take $\Delta = 0.02$ for the recommended parameters of Kyber; this gives a decryption failure probability $\delta' = 2^{-137}$, instead of 2^{-164} originally (see Table 8). We argue in Section 8.4 that Kyber remains secure with this increased decryption failure probability. We provide in Table 9 the value of the register size ℓ as a function of the number of shares n for $\Delta = 0.02$, according to Condition (12).

	n	k	q	η_1	η_2	(d_u, d_v)	δ	δ'
Kyber768	256	3	3329	2	2	(10,4)	2^{-164}	2^{-137}

Table 8. Parameter set for Kyber, with the original failure probability δ, and the failure probability δ' for $\Delta = 0.020$.

Application to Saber. For Saber, the original decryption failure probability is 2^{-136}, and with $\Delta = 0.02$ the failure probability becomes 2^{-112}. We can reach $\delta = 2^{-128}$ with $\Delta = 0.007$, but in that case there is no performance improvement compared to existing techniques. Therefore to guarantee $\delta \leq 2^{-128}$ we cannot use the techniques from this section.

n	2	3	4	5	6	7	8	9	10
ℓ	6	7	7	7	8	8	8	8	8

Table 9. Value of ℓ as a function of n with $\Delta = 0.02$ for Kyber.

8.4 Security impact for ring-LWE IND-CCA encryption

In this section we consider the security impact of increasing the decryption failure probability δ. Namely, as illustrated in Table 8, for the Kyber768 parameters the decryption failure probability becomes $\delta' = 2^{-137}$ instead of $\delta = 2^{-164}$, so we must explain why the Kyber scheme remains secure. For this we follow closely the analysis from [ABD+21, Section 5.5].

Classical security. We recall the CCA security of Kyber against classical adversaries, based on the Fujisaki-Okamoto transform, with a security bound that includes the decryption failure probability δ.

Theorem 7 (CCA security of Kyber [ABD+21]). *Suppose* XOF*, H, and G are random oracles. For any classical adversary $\mathcal{A}$ that makes at most q_{RO} many queries to random oracles* XOF*, H and G, there exist adversaries $\mathcal{B}$ and $\mathcal{C}$ of roughly the same running time as that of $\mathcal{A}$ such that* $\mathsf{Adv}^{\mathsf{cca}}_{\mathsf{Kyber.CCAKEM}} \leq 2\mathsf{Adv}^{\mathsf{mlwe}}_{k+1,k,\eta}(\mathcal{B}) + \mathsf{Adv}^{\mathsf{prf}}_{\mathsf{PRF}}(\mathcal{C}) + 4q_{RO} \cdot \delta$

We note that from the above security bound does not depend on the specific decryption algorithm used. This means that modifying the decryption algorithm (as we did in the previous section) does not invalidate the security proof of Kyber, as long as the decryption failure probability δ remains negligible. From the above security bound, with $\delta = 2^{-137}$, the best strategy to generate a decryption failure is to make $\simeq 2^{137}$ decryption or random oracle queries. This makes a classical attack completely unpractical.

Quantum security and failure boosting. In the quantum random oracle model, the security bound is non-tight and includes a term $q_{RO}^2 \cdot \delta$; see [ABD+21, Theorem 3]. Namely in the quantum setting the search for a m provoking a decryption failure can be quadratically accelerated using Grover's algorithm. In [ABD+21], the authors consider a failure boosting attack strategy that uses Grover's algorithm in an offline phase to search for a polynomial pair $(\mathbf{e}_1, \mathbf{r})$ with a larger norm, so that it is more likely to produce a decryption error. Below we use the same reasoning to estimate the quantum complexity of the attack, with decryption failure probability $\delta = 2^{-137}$ instead of 2^{-164}.

The polynomial pair $(\mathbf{e}_1, \mathbf{r})$ is seen as a vector in $\mathbb{Z}^{1536}$ distributed as a discrete Gaussian with standard deviation $\sigma = \sqrt{\eta_1/2} = 1$. We have that a m-dimensional vector $\mathbf{v}$ under such distribution satisfies for any $\kappa > 1$:

$$\Pr[\|\mathbf{v}\| > \kappa \cdot \sigma\sqrt{m}] < \kappa^m \cdot \exp(m(1-\kappa^2)/2)$$

Grover's algorithm is used to search this space with a quadratic speed-up, so with complexity $\kappa^{-m/2} \cdot \exp(m(\kappa^2-1)/4)$. In the second step, a decryption failure occurs if $\langle \mathbf{z}, \mathbf{v} \rangle$ is large enough for the secret vector $\mathbf{z}$. If $\mathbf{z}$ is distributed as a Gaussian with standard deviation σ', then for any λ, we have $\Pr[\langle \mathbf{z}, \mathbf{v} \rangle > \lambda\sigma'\|\mathbf{v}\|] \leq 2\exp(-\lambda^2/2)$. For a vector $\mathbf{v}$ without the failure boosting, we therefore have $\delta \simeq 2\exp(-\lambda^2/2)$, which gives $\lambda \simeq 13.8$ for $\delta = 2^{-137}$. Thanks to the failure boosting, we get a $\mathbf{v}$ whose norm is larger by a factor κ, so we can use $\lambda' = \lambda/\kappa$ instead of λ. The improved decryption failure probability after Grover's search then becomes $2\exp(-(\lambda/\kappa)^2/2)$, which gives a total complexity $\kappa^{-m/2} \cdot \exp(m(\kappa^2-1)/4 + (\lambda/\kappa)^2/2)$. For $\delta = 2^{-137}$, this is minimized for $\kappa = 1.1$, with total complexity 2^{124} (instead of 2^{150} for $\delta = 2^{-164}$). Therefore the attack remains completely unpractical. We refer to [ABD+21] for a discussion on the more recent attacks based on decryption failure [BS20,DRV20]; their overall running time for Kyber are no better than the above attack. In particular, the multi-target attack considered in [DGJ+19] is prevented in Kyber by hashing the public key pk into $\mathbf{r}$ and $\mathbf{e}_1$.

8.5 Comparison with existing techniques

We consider the computation of the threshold function th used in IND-CPA decryption for Saber and Kyber, and we provide a comparison of the operation count of our new technique (Algorithm 12) with existing techniques [BBE+18]. For Saber, we explain in Appendix D.1 how to compute the threshold function modulo 2^k based on the arithmetic to Boolean conversion algorithm from [CGV14]. For Kyber, we explain in Appendix D.2 how to compute the threshold function modulo q, based on the arithmetic modulo q to Boolean conversion from [BBE+18]. For our Algorithm 12, we use the register optimization (Algorithm 11) to perform the arithmetic modulo 2^ℓ to 1-bit Boolean conversion, according to the values of ℓ from Table 9. As in Section 6.3, we assume that a register operation takes 1 operation for 32-bit ($\ell = 5$), and $2^{\ell-5}$ operations for 2^ℓ-bit, for $\ell \geq 5$. We see in Table 10 that for Kyber, we obtain more than an order of magnitude improvement in IND-CPA decryption compared to [BBE+18].

	A mod $q \to$ 1-bit B	Security order t							
		1	2	3	4	5	6	8	9
Saber	[Gou01]	58							
	[CGV14]		366	667	1 286	1 979	2 731	4 767	6 051
Kyber	[BBE+18]	550	1 631	3 006	5 150	7 588	10 515	17 921	22 400
	Algorithm 12	26	117	220	355	1 026	1 421	2 403	2 990

Table 10. Operation count for arithmetic modulo q to 1-bit Boolean conversion algorithms, up to security order $t = 10$, with $n = t + 1$ shares, for Kyber and Saber (according to the values of ℓ from Table 9 for Algorithm 12), with $q = 3329$ for Kyber and $q = 2^{10}$ for Saber.

9 Binomial sampling and masked ring-LWE re-encryption

In this section, we show that our techniques enable to efficiently mask the re-encryption of ring-LWE encryption schemes. As recalled in Section 7.2, under a simplified version of the FO

transform for IND-CCA decryption, in the second step the message m is re-encrypted using error polynomials $(e_1, e_2, e_3) = H_1(m)$ to get a new ciphertext c'. To encode a Boolean masked message $m \in \{0,1\}$ as in (7), we can use our generic table-based conversion algorithm, with the function $f : \{0,1\} \to \mathbb{Z}_q$ with $f(x) = \lfloor q/2 \rceil \cdot x \bmod q$. In that case the complexity is $\mathcal{O}(n^2)$ as in [SPOG19].

Consider a single error e, which we write $e = H(m)$ for some hash function H; for simplicity we focus on a single component $e \in \mathbb{Z}$. The error e is actually computed using binomial sampling, with $(\alpha, \beta) = H(m)$ and then $e = h(\alpha) - h(\beta)$, where $\alpha, \beta \in \{0,1\}^k$ and h is the Hamming weight function. The message m is Boolean masked, and therefore the variables α and β are Boolean masked, while the error e must be arithmetically masked modulo q.

For masking the binomial sampling we must therefore mask the Hamming weight computation, with Boolean masking as input and arithmetic masking modulo q as output. Our approach is similar to [SPOG19]: we start from our 1-bit Boolean to arithmetic masking modulo q algorithm from Section 4.1 (with $k = 1$), and for $\alpha \in \{0,1\}^k$, the Hamming weight of α is computed as the sum of k independent 1-bit Boolean to arithmetic masking modulo q conversions. Starting from a Boolean masked message $m = m_1 \oplus \cdots \oplus m_n$, we then obtain an arithmetically masked ciphertext with n shares modulo q. Since our table-based approach has a similar level of efficiency as the technique from [SPOG19] (see Table 4 in Section 4.3 for a comparison), for the binomial sampling we obtain a similar level of efficiency as in [SPOG19], and an order of magnitude improvement compared to [BBE+18].

In the following, we describe in more details the technique to securely compute the Hamming weight and the binomial sampling, and we show how to perform masked IND-CPA encryption.

9.1 Masked Hamming weight computation

We consider the Hamming weight function $h : \{0,1\}^k \to \mathbb{Z}$ where $h(x)$ is the sum over $\mathbb{Z}$ of the bits of x, and the function $h_q : \{0,1\}^k \to \mathbb{Z}_q$ where this sum is computed modulo q, that is $h_q(x) = h(x) \bmod q$. Given as input $x_1, \ldots, x_n \in \{0,1\}^k$, our goal is to compute arithmetic shares $a_1, \ldots, a_n \in \mathbb{Z}_q$ such that:

$$a_1 + \cdots + a_n = h_q(x_1 \oplus \cdots \oplus x_n) \pmod q$$

The technique is similar to the optimized Boolean to arithmetic conversion algorithm from Section 4.2. We let $x = x_1 \oplus \cdots \oplus x_n$ and we write $x^{(j)}$ the j-th bit of x for $0 \le j < k$, which gives $h_q(x) = \sum_{j=0}^{k-1} x^{(j)}$. We also denote by $x_i^{(j)}$ the j-th bit of each share x_i. We obtain:

$$h_q(x) = \sum_{j=0}^{k-1} \bigoplus_{i=1}^{n} x_i^{(j)} \bmod q$$

We now perform an independent table-based Boolean to arithmetic conversion for each of the k variables $x^{(j)}$, namely we write for each $0 \le j < k$:

$$x^{(j)} = \bigoplus_{i=1}^{n} x_i^{(j)} = \sum_{i=1}^{n} y_i^{(j)} \pmod q$$

This gives:

$$h_q(x) = \sum_{j=0}^{k-1} \sum_{i=1}^{n} y_i^{(j)} = \sum_{i=1}^{n} \sum_{j=0}^{k-1} y_i^{(j)} \pmod q$$

and therefore letting $a_i := \sum_{j=0}^{k-1} y_i^{(j)}$ for all $1 \le i \le n$, we obtain $h_q(x_1 \oplus \cdots \oplus x_n) = a_1 + \cdots + a_n \pmod q$ as required. The algorithm is formally described in Algorithm 13 below. The complexity of the algorithm is $\mathcal{O}(k \cdot n^2)$. As for Algorithm 5, the $(n-1)$-SNI property of the algorithm follows from the $(n-1)$-SNI of each of the k independent table-based conversions.

Algorithm 13 Hamming weight

Input: $x_1, \ldots, x_n \in \{0,1\}^k$
Output: $a_1, \ldots, a_n \in \mathbb{Z}_q$ such that $a_1 + \cdots + a_n = h(x_1 \oplus \cdots \oplus x_n) \pmod q$

1: **for** $i = 1$ to n **do** $a_i \leftarrow 0$
2: **for** $j = 0$ to $k-1$ **do**
3: **for** $i = 1$ to n **do** $z_i \leftarrow (x_i \gg j)\ \&\ 1$
4: $(y_1^{(j)}, \ldots, y_n^{(j)}) \leftarrow \mathsf{BooleanToArithmetic}(1, z_1, \ldots, z_n)$
5: **for** $i = 1$ to n **do** $a_i \leftarrow a_i + y_i^{(j)} \bmod q$
6: **end for**
7: **return** $a_1, \ldots, a_n$

9.2 Application to binomial sampling and masked IND-CPA encryption

We consider the high-order masking of ring-LWE IND-CPA encryption, using error polynomials $(e_1, e_2, e_3) = H_1(m)$ from a message $m \in R$ with binary coefficients:

$$c_1 = a \cdot e_1 + e_2$$
$$c_2 = t \cdot e_1 + e_3 + \lfloor q/2 \rceil \cdot m$$

We are given as input a Boolean masked message $m = m_1 \oplus \cdots \oplus m_n \in R$ and we must output an arithmetically masked ciphertext modulo q. Applying our generic conversion algorithm with the function $f : \{0,1\} \to \mathbb{Z}_q$ with $f(x) = \lfloor q/2 \rceil \cdot x \bmod q$ on each coefficient separately, we obtain arithmetic shares $M_1, \ldots, M_n \in R_q$ such that:

$$m \cdot \lfloor q/2 \rceil = \sum_{i=1}^{n} M_i \pmod q$$

Similarly, each component $e \in \mathbb{Z}$ of the error polynomials e_1, e_2, e_3 is equal to $e = h_q(\alpha) - h_q(\beta) \bmod q$, where $\alpha, \beta \in \{0,1\}^k$ and h_q is the Hamming weight function modulo q. Starting from n-shared Boolean masking of α and β, we can therefore apply Algorithm 13 to generate n arithmetic shares for e modulo q. Eventually, we obtain arithmetically masked error polynomials $E_{ji} \in \mathcal{R}_q$ for $j = 1, 2, 3$ such that

$$e_j = \sum_{i=1}^{n} E_{ji} \pmod q$$

Finally, we can compute the n shares of the ciphertext:

$$c_{1,i} = a \cdot E_{1,i} + E_{2,i}$$
$$c_{2,i} = t \cdot E_{1,i} + E_{3,i} + \lfloor q/2 \rceil \cdot M_i$$

and we have $\sum_{i=1}^{n} c_{1,i} = c_1 \pmod q$ and $\sum_{i=1}^{n} c_{2,i} = c_2 \pmod q$ as required. Therefore we have obtained a masked ciphertext with n shares modulo q. The complexity is $\mathcal{O}(n^2)$ for n shares.

10 Practical implementation

We have performed a plain C implementation of our techniques, and of [CGV14,BBE+18] and [SPOG19] for comparison. In the following, we provide tables containing the average cycle count for each gadget over 1 000 000 executions, running on an `Intel(R) Core(TM) i7-1065G7 CPU @ 1.30GHz` processor of a laptop, for security orders $1 \leq t \leq 9$. Randomness generation has been performed using a simple xorshift PRNG. Still we provide in Table 15 the number of randoms needed for all considered techniques to fairly highlight and compare the randomness usage.

10.1 Conversion from 1-bit Boolean to arithmetic modulo q

The conversion from 1-bit Boolean to arithmetic modulo q is a core component of the IND-CPA encryption since it is used to both encode the message m and to protect the binomial sampling (see Section 9.2). Table 11 shows a comparison between our technique depicted in Algorithm 3 (BooleanToArithmetic) and [SPOG19]. We only compare with [SPOG19] since according to Table 4, [BBE+18] is much less efficient. We see that we get a similar level of efficiency, as it was also the case in the operation count from Table 4.

1-bit B → A	Security order t								
	1	2	3	4	5	6	7	8	9
[SPOG19]	61	142	250	408	520	687	882	1 133	1 354
BooleanToArithmetic	59	98	207	347	524	749	962	1 241	1 618

Table 11. Number of cycles to perform a 1-bit Boolean to arithmetic modulo q conversion for $q = 3329$.

10.2 Arithmetic shift by ℓ bits

We have implemented the Shift1 algorithm (Algorithm 6), which requires a table of size $B = n \cdot (2^\ell - 1) + 1$ rows for securely computing the carry value $0 \leq c \leq c_{max}$ with $c_{max} = \lfloor n \cdot (2^\ell - 1)/2^\ell \rfloor$. Recall that in Algorithm 6 the carry is arithmetically masked modulo $2^{k-\ell}$, see Equation (2). Therefore, in principle one cannot apply the register optimization from Section 6.3, since such optimization requires a Boolean masked output. However we used the following trick. Instead of storing the carry c directly in arithmetic masking as in Algorithm 6, we first use a Boolean masking for storing c. This requires n Boolean masks as output, with $\lceil \log_2 c_{max} \rceil$ bits each to represent the carry. With a Boolean masking as output, we can now apply the arithmetic to Boolean conversion with register optimization from Section 6.3, with registers of size $B \cdot \lceil \log_2 c_{max} \rceil$ bits. Eventually we convert the carry from Boolean to arithmetic masking modulo $2^{k-\ell}$, using the BAopti algorithm (Algorithm 5). Therefore, in Algorithm 6 the arithmetic refreshes are replaced by Boolean refreshes (using the register optimization from Section 6.3), and a Boolean to arithmetic conversion is performed before Line 10. With $\ell = 1$, we can use 8-bit registers for $n = 2$, 16-bit registers for $n \leq 7$, and 64-bit registers for $n \leq 15$. We also considered $\ell = 3$ as in Saber, with 16-bit registers for $n = 2$, and 64-bit registers for $n = 3, 4$. For other values of ℓ, we can iterate multiple Shift1 with $\ell = 1$.

We provide the implementation results in Table 12 with input arithmetic values modulo 2^k, with $k = 13$ as in Saber. Recall that the [CGV14] technique is not sensitive to the number of

shifts ℓ. We see that for $\ell = 1$ and $\ell = 3$ our technique outperforms [CGV14], especially for small security orders. For $\ell = 3$, we see a significant gap in performances between order 3 and order 4. Namely for order 3 ($n = 4$) we can store the carry in 64-bit registers as explained above, while for higher order we simply use 3 iterations with $\ell = 1$.

A → A shift, $k = 13$	Security order t								
	1	2	3	4	5	6	7	8	9
[CGV14]	270	1183	1 743	3 176	4 819	6 107	7 610	10 108	12 661
Shift1 ($\ell = 1$)	15	223	429	686	1104	1 567	2 967	3 822	4 879
Shift1 ($\ell = 3$)	16	230	443	2 087	3 339	4 689	9 006	11 613	15 053

Table 12. Number of cycles to perform an arithmetic to arithmetic shift by ℓ positions where $\ell = 1$ and $\ell = 3$, with input arithmetic values modulo 2^k where $k = 13$.

10.3 Arithmetic modulo 2^k to k-bit Boolean conversion

We consider the classical arithmetic modulo 2^k to k-bit Boolean conversion, for small k, which is depicted in Algorithm 8 (ArithmeticToBoolean). This conversion is a building block of the optimized arithmetic to Boolean conversion of arbitrary size of Section 6.2. Indeed, Algorithm 10 relies on a smaller conversion for each block of ℓ bits. For small values of k, as previously we can use the register optimization from Section 6.3 with a register size of $k \cdot 2^k$ bits. We can work with the conversion $\mathbb{Z}_{16} \to \{0,1\}^4$, *i.e.* with $k = 4$ since the table fits in a 64-bit register. Indeed, using an uint64_t to store each share of the table, the conversion simply consists in cyclically shifting those integers and XORing them with random 64-bit values. Table 13 shows that this technique for small k is more efficient than the conversion of [CGV14]. We recall that for large k, it will however not scale well since the size of the table is exponential in the number of bits converted. Namely, for large k, one should use our optimized algorithm from Section 6.2.

A → B from $\mathbb{Z}_{16}$ **to** $\{0,1\}^4$	Security order t								
	1	2	3	4	5	6	7	8	9
[CGV14]	9	158	228	460	688	850	1 066	1 392	1 777
ArithmeticToBoolean	10	32	71	108	159	226	295	390	511

Table 13. Number of cycles to perform an arithmetic modulo 2^k to k-bit Boolean conversion with $k = 4$.

10.4 Threshold decryption for Kyber

We have implemented the threshold decryption function th for Kyber and Saber, which is a core component of the IND-CPA decryption. For Kyber we have used the approach described in Section 8.2 with Algorithm 12, where we first perform a modulus switching to a smaller modulus 2^ℓ, and then compute the threshold function $f : 2^\ell \to \{0,1\}$ from arithmetic to Boolean masking. We see in Table 9 that the function must be computed for $\ell = 6, 7, 8$ for a number of shares ≤ 11. For $\ell = 6$, we can therefore use the register optimization from Section 6.3, and the table with $2^\ell = 64$ rows can fit in a physical register or at least be smoothly managed by the compiler using the appropriate data type. The cases $\ell = 7, 8$ are somewhat trickier. Even if some architecture

might support 128-bit or 256-bit integers, these widths are less common. In those cases, we use two or four 64-bit registers and simulate the shift on 128 or 256 bits with 64-bit instructions. The results in Table 14 shows that for Kyber our approach significantly outperforms [BBE+18], and becomes even faster than in Saber.

	A mod q →	Security order t								
	1-bit B	1	2	3	4	5	6	7	8	9
Saber	[CGV14]	24	335	507	994	1 532	1 956	2 259	4 281	7 120
Kyber	[BBE+18]	831	2 027	3 276	8 003	13 961	20 424	27 165	34 834	42 772
	ThresholdAtoB	20	88	155	236	870	466	1 367	1 704	2 240

Table 14. Number of cycles to perform a threshold decryption including a conversion from arithmetic modulo q to 1-bit Boolean, for Kyber (with $q = 3329$) and Saber (with $q = 2^{10}$). For our TresholdAtoB algorithm (Algorithm 12), this is according to the values of ℓ from Table 9.

10.5 Randomness usage

Due to the refresh gadgets, the practical performance of masking schemes is strongly impacted by the speed of the RNG. Table 15 shows the number of RNG calls outputting 32-bit values for each execution of the gadgets considered in the previous sections. We assume that the RNG always outputs exactly 32 bits, which means that a fresh 16-bit value also counts for one call whereas a fresh 64-bit value counts for two.

	Security order t								
	1	2	3	4	5	6	7	8	9
1-bit B. to A. [SPOG19]	3	8	16	25	37	51	68	85	105
1-bit B. to A.	3	10	21	36	55	78	105	136	171
Shift of 1 position [CGV14]	17	125	249	485	764	1 080	1 439	1 931	2 466
Shift of 1 position	3	26	54	92	140	198	427	552	693
Shift of 3 positions [CGV14]	15	113	225	439	692	978	1 303	1 749	2 234
Shift of 3 positions	3	32	66	276	420	594	1 281	1 656	2 079
$\mathbb{Z}_{16} \to \{0,1\}^4$ [CGV14]	1	15	30	60	96	135	180	242	310
$\mathbb{Z}_{16} \to \{0,1\}^4$	3	10	21	36	55	78	105	136	171
Saber decrypt [CGV14]	1	18	42	86	138	225	300	407	523
Saber decrypt	2	6	24	40	60	150	203	264	330
Kyber decrypt [BBE+18]	22	91	182	370	564	804	1 072	1 576	1 995
Kyber decrypt	2	6	12	40	60	85	112	264	330

Table 15. Number of 32-bit randoms needed for one execution of each gadget.

11 Conclusion

We have described a new high-order conversion algorithm between Boolean and arithmetic masking, based on a generalization of the table recomputation countermeasure from [Cor14]. For classical k-bit to k-bit conversions, the new algorithm offers a similar level of efficiency as in [CGV14]. For 1-bit Boolean to arithmetic modulo q conversion, the new algorithm offers a

similar level of efficiency as in [SPOG19]. For the computation of a threshold function from arithmetic to 1-bit Boolean masking (as used in the IND-CPA decryption of M-LWE encryption schemes such as Kyber and Saber), we obtain an order of magnitude improvement compared to the state of the art, thanks to a new modulus switching technique over arithmetic shares, and an optimization of the table recomputation in registers. This was confirmed by the results of a practical implementation.

We think that the main advantage of our high-order table-based conversion algorithm is its flexibility: we can start from any masking as input (either Boolean, or arithmetic modulo 2^k or any q), compute any function f (for example threshold or shift), and obtain any masking as output, with a good level of efficiency, and with a simpler implementation than with existing techniques.

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.

[ADPS16] Erdem Alkim, Léo Ducas, Thomas Pöppelmann, and Peter Schwabe. Post-quantum key exchange - A new hope. In *25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016*, pages 327–343, 2016.

[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.

[BCZ18] Luk Bettale, Jean-Sébastien Coron, and Rina Zeitoun. Improved high-order conversion from boolean to arithmetic masking. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2018(2):22–45, 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.

[BDK+20] Michiel Van Beirendonck, Jan-Pieter D'Anvers, Angshuman Karmakar, Josep Balasch, and Ingrid Verbauwhede. A side-channel resistant implementation of SABER. *IACR Cryptol. ePrint Arch.*, 2020:733, 2020.

[BDV21] Michiel Van Beirendonck, Jan-Pieter D'Anvers, and Ingrid Verbauwhede. Analysis and comparison of table-based arithmetic to boolean masking. Cryptology ePrint Archive, Report 2021/067, 2021. `https://eprint.iacr.org/2021/067`.

[BGR+21] Joppe W. Bos, Marc Gourjon, Joost Renes, Tobias Schneider, and Christine van Vredendaal. Masking kyber: First- and higher-order implementations. Cryptology ePrint Archive, Report 2021/483, 2021. `https://eprint.iacr.org/2021/483`.

[BHLY16] Leon Groot Bruinderink, Andreas Hülsing, Tanja Lange, and Yuval Yarom. Flush, gauss, and reload - A cache attack on the BLISS lattice-based signature scheme. In *Proceedings of CHES 2016*, pages 323–345, 2016.

[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.

[BS20] Nina Bindel and John M. Schanck. Decryption failure is more likely after success. In *Post-Quantum Cryptography - 11th International Conference, PQCrypto 2020, Paris, France, April 15-17, 2020, Proceedings*, pages 206–225, 2020.

[CGTV15] Jean-Sébastien Coron, Johann Großschädl, Mehdi Tibouchi, and Praveen Kumar Vadnala. Conversion from arithmetic to boolean masking with logarithmic complexity. In *Proceedings of FSE 2015*, pages 130–149, 2015.

[CGV14] Jean-Sébastien Coron, Johann Großschädl, and Praveen Kumar Vadnala. Secure conversion between boolean and arithmetic masking of any order. In *Proceedings of CHES 2014*, pages 188–205, 2014.

[CJRR99] Suresh Chari, Charanjit S. Jutla, Josyula R. Rao, and Pankaj Rohatgi. Towards sound approaches to counteract power-analysis attacks. In *CRYPTO*, 1999.

[Cor14] Jean-Sébastien Coron. Higher order masking of look-up tables. In *Proceedings of EUROCRYPT 2014*, pages 441–458, 2014.

[Cor17] Jean-Sébastien Coron. High-order conversion from boolean to arithmetic masking. In *Proceedings of CHES 2017*, pages 93–114, 2017. Full version available at `http://eprint.iacr.org/2017/252`.

[CRZ18] Jean-Sébastien Coron, Franck Rondepierre, and Rina Zeitoun. High order masking of look-up tables with common shares. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2018(1):40–72, 2018.

[CT03] Jean-Sébastien Coron and Alexei Tchulkine. A new algorithm for switching from arithmetic to boolean masking. In *Proceedings of CHES 2003*, pages 89–97, 2003.

[DDF14] Alexandre Duc, Stefan Dziembowski, and Sebastian Faust. Unifying leakage models: From probing attacks to noisy leakage. In *Advances in Cryptology - EUROCRYPT 2014 - Proceedings*, pages 423–440, 2014.

[Deb12] Blandine Debraize. Efficient and provably secure methods for switching from arithmetic to boolean masking. In Emmanuel Prouff and Patrick Schaumont, editors, *Proceedings of CHES 2012*, volume 7428 of *Lecture Notes in Computer Science*, pages 107–121. Springer, 2012.

[DGJ+19] Jan-Pieter D'Anvers, Qian Guo, Thomas Johansson, Alexander Nilsson, Frederik Vercauteren, and Ingrid Verbauwhede. Decryption failure attacks on IND-CCA secure lattice-based schemes. 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 565–598, 2019.

[DNR04] Cynthia Dwork, Moni Naor, and Omer Reingold. Immunizing encryption schemes from decryption errors. In *Advances in Cryptology - EUROCRYPT 2004, International Conference on the Theory and Applications of Cryptographic Techniques, Interlaken, Switzerland, May 2-6, 2004, Proceedings*, pages 342–360, 2004.

[DRV20] Jan-Pieter D'Anvers, Mélissa Rossi, and Fernando Virdia. (one) failure is not an option: Bootstrapping the search for failures in lattice-based encryption schemes. In *Advances in Cryptology - EUROCRYPT 2020 - Part III*, pages 3–33, 2020.

[EFGT17] Thomas Espitau, Pierre-Alain Fouque, Benoît Gérard, and Mehdi Tibouchi. Side-channel attacks on BLISS lattice-based signatures: Exploiting branch tracing against strongswan and electromagnetic emanations in microcontrollers. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, pages 1857–1874, 2017.

[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.

[Gou01] Louis Goubin. A sound method for switching between boolean and arithmetic masking. In Çetin Kaya Koç, David Naccache, and Christof Paar, editors, *Cryptographic Hardware and Embedded Systems - CHES 2001, Third International Workshop, Paris, France, May 14-16, 2001, Proceedings*, volume 2162 of *Lecture Notes in Computer Science*, pages 3–15. Springer, 2001.

[GR19] François Gérard and Mélissa Rossi. An efficient and provable masked implementation of qtesla. In *Smart Card Research and Advanced Applications - 18th International Conference, CARDIS 2019, Prague, Czech Republic, November 11-13, 2019, Revised Selected Papers*, pages 74–91, 2019.

[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.

[Koc96] Paul C. Kocher. Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems. In *Advances in Cryptology - CRYPTO '96, 16th Annual International Cryptology Conference, Santa Barbara, California, USA, August 18-22, 1996, Proceedings*, pages 104–113, 1996.

[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.

[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.

[MGTF19] Vincent Migliore, Benoît Gérard, Mehdi Tibouchi, and Pierre-Alain Fouque. Masking dilithium - efficient implementation and side-channel evaluation. In *Applied Cryptography and Network Security - 17th International Conference, ACNS 2019, Bogota, Colombia, June 5-7, 2019, Proceedings*, pages 344–362, 2019.

[OMHT06] Elisabeth Oswald, Stefan Mangard, Christoph Herbst, and Stefan Tillich. Practical second-order DPA attacks for masked smart card implementations of block ciphers. In David Pointcheval, editor, *Proceedings of CT-RSA 2006*, volume 3860 of *Lecture Notes in Computer Science*, pages 192–207. Springer, 2006.

[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.

[PG13] Thomas Pöppelmann and Tim Güneysu. Towards practical lattice-based public-key encryption on reconfigurable hardware. In *Selected Areas in Cryptography - SAC 2013 - 20th International Conference, Burnaby, BC, Canada, August 14-16, 2013, Revised Selected Papers*, pages 68–85, 2013.

[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.

[RRVV15] Oscar Reparaz, Sujoy Sinha Roy, Frederik Vercauteren, and Ingrid Verbauwhede. A masked ring-lwe implementation. In *Cryptographic Hardware and Embedded Systems - CHES 2015 - 17th International Workshop, Saint-Malo, France, September 13-16, 2015, Proceedings*, pages 683–702, 2015.

[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.

[TE15] Mostafa Taha and Thomas Eisenbarth. Implementation attacks on post-quantum cryptographic schemes. *IACR Cryptol. ePrint Arch.*, 2015:1083, 2015.

[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 Proof of Theorem 1

We use the following Lemma, whose proof is straightforward and therefore omitted.

Lemma 2. *Let $(y_i)_{1\leq i\leq n}$ be the input and let $(z_i)_{1\leq i\leq n}$ be the output of* $\mathsf{Refresh}_{\mathsf{H}}$*. Any subset of $n-1$ output variables z_i is uniformly and independently distributed in H.*

The proof of Theorem 1 is relatively similar to the proof of the table-based countermeasure in [Cor14]. Given $u \in G$, we denote by $T(u)[j]$ and $T'(u)[j]$ the j-th component of the vectors $T(u)$ and $T'(u)$ respectively, for $1 \leq j \leq n$. We denote by *Part* i the computation performed within the main for loop, that is between lines 2 and 5 of Algorithm 1, and by *Part* n the computation performed at line 6. We describe hereafter the construction of two index sets I and J, both initially empty.

- For every probed input variable x_i or any intermediate variable $u + x_i$ (for any $1 \leq i \leq n$), we add i to I.
- For every probed intermediate variable $T'(u)[j] = T(u + x_i)[j]$, or r_j or y_j in $\mathsf{Refresh}_H$ (for any $1 \leq i \leq n$) in Part i, or $T(u)[j]$ in line 4 of Part i, we add i to I and j to J.
- For every output y_j such that $j \in O$, we add j to J.

Since for every probed variable we added at most one index in I, we have $|I| \leq t_1$ as required. Similarly for J we must have $|J| \leq |O| + t_1 < n$.

We now show that any set of t_1 variables and the output shares $y_{|O}$ can be perfectly simulated from $x_{|I}$. This is clear for the probed input variables x_i and intermediate variables $u + x_i$ (for any $1 \leq i \leq n$ and all $u \in G$), since by construction $i \in I$. It remains to perfectly simulate all probed variables of the form $T(u)[j]$, $T'(u)[j]$ and $T(u + x_i)[j]$, including the output variables $y_{|O}$. We proceed by induction on i. Namely we show that at the beginning of each part i, we can perfectly simulate all variables $T(u)[j]$ for all $j \in J$ and all $u \in G$. This holds for the case $i = 1$, since at the beginning of Part 1, the vector $T(u) = \big(f(u), 0, \ldots, 0\big)$ is publicly known. At the beginning of Part i, we distinguish two cases:

Case $i \in I$. If $i \in I$ then knowing x_i we can perfectly simulate all intermediate variables with column index $j \in J$ in Part i, as knowing x_i we can propagate the simulation for all variables with column index j and perfectly simulate $T(u + x_i)[j]$, $T'(u)[j]$ and the resulting $T(u)[j]$ at Line 4, and similarly the variables y_j at Line 6 if $i = n$; in particular the r_j variables within $\mathsf{Refresh}_H$ are simulated exactly as in the $\mathsf{Refresh}_H$ procedure.

Case $i \notin I$. If $i \notin I$ then no variable in Part i has been probed, including variables in $\mathsf{Refresh}_H$. Since $|J| < n$, using Lemma 2 we can therefore perfectly simulate all intermediate variables $T(u)[j]$ for $j \in J$ and $u \in G$ at the output of $\mathsf{Refresh}_H$ at Line 4, or similarly all y_j for $j \in J$ at the output of $\mathsf{Refresh}_H$ at Line 6 when $i = n$, simply by generating uniform and independent values.

As a conclusion, we have shown that for all i the induction step is verified, which means that all $T(u)[j]$, $T'(u)[j]$ and $T'(u + x_i)[j]$ for $j \in J$ can be perfectly simulated from $x_{|I}$, including the output variables $y_{|O}$. Therefore all probes can also be perfectly simulated. This terminates the proof of Theorem 1.

B Decryption failure probability in Kyber

For Kyber, the decryption failure probability is computed by the following lines in the Python file Kyber_failure.py, which is publicly available in:

https://github.com/pq-crystals/security-estimates/

```
def p2_cyclotomic_error_probability(ps):
    F = p2_cyclotomic_final_error_distribution(ps)
    proba = tail_probability(F, ps.q/4)
    return F, ps.n*proba
```

Therefore to compute our new decryption error probability, it suffices to replace the $q/4$ bound by $q \cdot (1/4 - \Delta)$:

```
    proba = tail_probability(F, ps.q*(1/4-Delta))
```

For Kyber with the recommended parameters, the original decryption failure with $\Delta = 0$ is 2^{-164}. By running the script with $\Delta = 0.002$, we obtain a failure probability of 2^{-137}.

C Operation count in previous work

[Gou01]. The first-order Boolean to arithmetic conversion from [Gou01] requires 7 operations. The first-order arithmetic to Boolean conversion from [Gou01] requires $5k + 5$ operations. The high-order Boolean to arithmetic conversion from [BCZ18] requires $10 \cdot 2^n - 6 \cdot n - 13$ operations.

[SPOG19]. The k-bit Boolean to arithmetic conversion from [SPOG19] has number of operations $T_{\mathsf{SecB2Aq}}(n, k) = 9kn^2/2 + 5kn/2 - 2n - 3k$.

Arithmetic modulo 2^k to Boolean conversion [CGV14]. The complexity of SecAnd algorithm in [CGV14, Algorithm 1] is the same as for the And gadget in [ISW03], and is given by:

$$T_{\mathsf{SecAnd}}(n) = 5 \cdot \frac{n(n-1)}{2} + n^2 = \frac{7n^2}{2} - \frac{5n}{2}$$

For the SecAdd algorithm in [CGV14], this gives:

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

The complexity of the Expand algorithm is $2\lfloor n/2 \rfloor$. For the arithmetic modulo 2^k to k-bit Boolean conversion, we have the recursion:

$$T_{\mathsf{AB}}(n, k) = T_{\mathsf{AB}}(\lfloor n/2 \rfloor, k) + T_{\mathsf{AB}}(\lceil n/2 \rceil, k) + 4\lfloor n/2 \rfloor + T_{\mathsf{SecAdd}}(n, k)$$

and $T_{\mathsf{AB}}(1, k) = 0$. For $n = 2$ we can use the first-order secure conversion algorithm from [Gou01], with $5k + 5$ operations, so we take $T_{\mathsf{AB}}(2, k) = 5k + 5$.

Arithmetic modulo p to Boolean conversion [BBE+18]. For the secure addition modulo p, we have:

$$T_{\mathsf{SecAddModp}}(n, k) = (2k + 2)(T_{\mathsf{SecAnd}}(n) + 2n) + \frac{3n^2}{2} - \frac{n}{2} + 2$$

For the Arithmetic mod p to Boolean conversion, we have the same recursion as previously:

$$T_{\mathsf{ABModp}}(n, k) = T_{\mathsf{ABmodp}}(\lfloor n/2 \rfloor, k) + T_{\mathsf{ABmodp}}(\lceil n/2 \rceil, k) + 4\lfloor n/2 \rfloor + T_{\mathsf{SecAddModp}}(n, k)$$

and $T_{\mathsf{ABModp}}(1, k) = 0$.

Boolean to arithmetic modulo 2^k conversion [CGV14]. We have:

$$T_{\mathsf{BA}}(n,k) = T_{\mathsf{AB}}(n,k) + T_{\mathsf{SecAdd}}(n,k) + 3n^2 - 3$$

High-order Boolean to arithmetic modulo p conversion [BBE+18]. We have as previously:

$$T_{\mathsf{BAModp}}(n,k) = T_{\mathsf{ABModp}}(n,k) + T_{\mathsf{SecAddModp}}(n,k) + 3n^2 - 3$$

D Computing the threshold function

D.1 The mod 2^k case

We show how to compute a threshold function $\mathsf{th} : \mathbb{Z}_{2^k} \to \{0,1\}$ where $\mathsf{th}(x) = 0$ if $x \in [0, 2^{k-1} - 1]$ and $\mathsf{th}(x) = 1$ otherwise, as used in Saber with $k = 10$. Note that $\mathsf{th}(x)$ is equal to the most significant bit of the k-bit representation of x. Starting from $x = x_1 + \cdots + x_n \pmod{2^k}$, we perform an arithmetic to Boolean conversion of $(x_i)_{1 \le i \le n} \in \mathbb{Z}_{2^k}$ into $(z_i)_{1 \le i \le n} \in \{0,1\}^k$. We then let $b_i \in \{0,1\}$ be the most significant bit of z_i for $1 \le i \le n$. We obtain $b_1 \oplus \cdots \oplus b_n = \mathsf{th}(x_1 + \cdots + x_n)$ as required. The operation count is therefore:

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

D.2 The mod q case

We consider an integer q such that $q \equiv 1 \pmod 4$, as in Kyber with $q = 3329$. We show how to compute the threshold function $\mathsf{th} : \mathbb{Z}_q \to \{0,1\}$ where $\mathsf{th}(x) = 0$ if $x \in (-q/4, q/4)$ and $\mathsf{th}(x) = 1$ otherwise, where x is represented in $[-(q+1)/2, (q-1)/2]$. We first compute $y = x + (q-1)/4 \in \mathbb{Z}_q$ and we consider the function $\mathsf{th}'(y) = \mathsf{th}(x)$. We obtain $\mathsf{th}'(y) = 0$ if $y \in [0, (q-1)/2]$ and $\mathsf{th}'(y) = 1$ otherwise, where y is represented in $[0, q-1]$.

We consider k such that $q < 2^k$. We now consider y over $\mathbb{Z}$ with $0 \le y < q < 2^k$. We let $z = y - (q+1)/2 \bmod 2^k$. If $0 \le y \le (q-1)/2$, then $-2^{k-1} \le -(q+1)/2 \le y - (q+1)/2 < 0$. In this case we have $z = y - (q+1)/2 + 2^k$, which gives $2^{k-1} \le z < 2^k$, and therefore the most significant bit of z is 1. Otherwise, if $(q+1)/2 \le y < q$, then $0 \le y - (q+1)/2 < (q-1)/2 \le 2^{k-1}$, and therefore $z = y - (q+1)/2$, which gives $0 \le z < 2^{k-1}$ and therefore the most significant bit of z is 0. Letting b be the most significant bit of z, we have $\mathsf{th}(x) = \neg b$.

Starting from $x = x_1 + \cdots + x_n \pmod q$, we compute $y_1 = x_1 + (q-1)/4 \in \mathbb{Z}_q$ and $y_i = x_i$ for $2 \le i \le n$. We then perform an arithmetic modulo q to Boolean conversion of $(y_i)_{1 \le i \le n} \in \mathbb{Z}_q$ into $(u_i)_{1 \le i \le n} \in \{0,1\}^k$. We then perform a SecAdd between $(u_i)_{1 \le i \le n}$ and $(v_i)_{1 \le i \le n}$ with $v_1 = 2^k - (q+1)/2$ and $v_i = 0$ for $2 \le i \le n$. We obtain $(z_i)_{1 \le i \le n}$ and we let b_i be the most significant bit of z_i for $1 \le i \le n$. We let $b_1 \leftarrow \neg b_1$. Eventually we obtain $b_1 \oplus \cdots \oplus b_n = \mathsf{th}(x_1 + \cdots + x_n)$ as required. The operation count is therefore:

$$T_{\mathsf{thModp}}(n,k) = T_{\mathsf{ABModp}}(n,k) + T_{\mathsf{SecAdd}}(n,k) + n + 2$$