Masked compression for ML-KEM

Guilhèm Assael ✉[1] and Gilles Van Assche[2]

[1] STMicroelectronics, Rousset, France
guilhem.assael@st.com
[2] STMicroelectronics, Diegem, Belgium
gilles.vanassche@st.com

Abstract. This paper proposes a new gadget to compute securely at any masking order the compression step in the ML-KEM post-quantum key encapsulation mechanism. The proposed solution is actually more general and can rescale from or to any range size, making ML-KEM's compression a particular case. It works as a share-wise approximation step followed by a nonlinear correction step. The gadget has a complexity that compares favorably with the state of the art, and most importantly it avoids higher-precision arithmetic as in some existing solutions. As a result, the gadget's data width is independent of the masking order, allowing one to reuse the secure-addition and masking-conversion gadgets already employed in other parts of a secure implementation of ML-KEM.

Keywords: ML-KEM ciphertext compression · Masking · Side-channel countermeasures.

1 Introduction

Research in quantum computing has shown that traditional public-key cryptographic algorithms, such as RSA, Diffie-Hellman or elliptic curve-based schemes, can be broken in polynomial time [25]. In the last decade, there has been a growing interest for so-called post-quantum cryptography, that is, cryptographic schemes that resist quantum attacks. Standardized by NIST in 2024 after running an open competition, ML-KEM is a prominent example of post-quantum algorithm [15], and more specifically a key encapsulation mechanism, allowing the users to establish a common secret.

As a lattice-based scheme, the implementation of ML-KEM raises new challenges, especially on embedded systems where side-channel and fault attacks are a concern [20]. Protecting against electrical and/or electromagnetic side-channel attacks typically involves masking, that is, representing internal variables in a number of shares that do not carry information on secret values when taken individually. Masking has been extensively studied [11, 19], and yet ML-KEM remains relatively difficult to protect, stemming from the relatively heterogeneous set of operations it performs, involving different representations such as integers modulo the prime $q_{\mathrm{K}} = 3329$ or a power of two, and from the use of a variant of the Fujisaki-Okamoto (FO) transform [17]. The latter requires a secure way to compare the re-encrypted ciphertext with the received one.

In this paper, we focus on this comparison, and more specifically on the fact that the ciphertexts are compressed, that is, they are scaled from $\mathbb{Z}/(q_{\mathrm{K}}\mathbb{Z})$ to $\mathbb{Z}/(2^d\mathbb{Z})$ for $d \in \{4, 5, 10, 11\}$. Because of this compression, the comparison requires either a masked ciphertext-compression operation, or an ad-hoc masked comparison operation that can handle one ciphertext being masked but uncompressed, while the other is compressed but unprotected. The second method, chosen for instance by Bos *et al.* [6], lacks in generality: In particular, it does not allow to mask encapsulation fully. The first method, on the other hand, has not received much attention: Apparently, a single solution for its implementation, due to Fritzmann *et al.* [16], has been proposed in the literature, and has been better specified by Coron *et al.* [12] in the case of high-order masking. A shortcoming of this solution is its reliance on high-precision arithmetic and high bit-width masking conversions. The other masked operations of ML-KEM use 12-bit arithmetic and 12-bit masking conversions; in contrast, Fritzmann *et al.* call for integer division with a 24-bit quotient (constituted of 11 integral bits and 13 fractional bits), and a 24-bit arithmetic-to-Boolean (A2B) masking conversion.

Our contribution In this paper, we present a different solution for the masking of ML-KEM compression[3]. Furthermore, we show that our solution can be generalized to any similar rounded rescaling operation: we securely implement the Rescale function

$$\mathrm{Rescale}_{q,q',r}(x) = \left\lfloor \frac{xq' + r}{q} \right\rfloor \bmod q' ,$$

where q and q' can be any integers including $q = q_{\mathrm{K}}$ and $q' = 2^d$ for ML-KEM. We notice that, at least when $r = 0$, $\mathrm{Rescale}(x)$ is close to a linear function in x, that is, $\mathrm{Rescale}(x+y) \approx \mathrm{Rescale}(x) + \mathrm{Rescale}(y)$. Roughly, our strategy consists in evaluating $\mathrm{Rescale}(x)$ on each arithmetic share individually and providing a secure way to compute the correction terms. The main benefit of our solution is that it does not rely on higher-precision arithmetic or on larger-width masking conversions than the rest of the masked scheme.

Organization of the paper After recalling the background notions, security model and previous works in Section 2, we present in Section 3 the problem we set out to solve. This section also presents the solution, including references to the existing gadgets we rely on, starting from the simpler case of first-order masking, then extending to high-order masking. In Section 4, we prove the correctness and security of our solution, propose some optimizations, and compare our approach with the state of the art. Finally, we conclude in Section 5.

2 Preliminaries

We start by introducing some background: after specifying the notations that we employ, we recall the security notion of Probe-Isolating Non Interference

[3] Our solution is patented in [3]; a preliminary version of this work is available in [1].

(PINI) and its composability properties. We then present ML-KEM ciphertext compression, together with a slightly more generic operation that we call *integer rescaling*. We then review the solutions that have been proposed in the literature to either mask this operation, or securely dispose with the need to mask it.

2.1 Countering vertical side-channel attacks with masking

Since the foundational work of Chari *et al.* [11], *masking* has kept developing as a powerful countermeasure against vertical side-channel attacks. The principle of this countermeasure, formerly used for other purposes under the name of *secret sharing*, is to divide a cryptographic secret into several parts, or *shares*, such that no secret information can be deduced from a subset of the shares, but knowing the full set of shares still allows to perform the desired operation.

Practically, given a secret x element of some group $(G, \star)$, an n-share sharing of x is a list $(x^{\mathbf{0}}, \ldots, x^{\boldsymbol{n-1}}) \in G^n$, such that $x = x^{\mathbf{0}} \star \cdots \star x^{\boldsymbol{n-1}}$. The masked implementation of a particular function f is a process that securely computes a sharing of $f(x)$ from the provided sharing of x. Such an implementation is known as a *masked gadget*.

Designing and verifying the masked implementation of a cryptographic operation is a difficult task. Thus, a compositional approach is often used, whereby complex gadgets are constructed as the *composition* of more elementary gadgets whose security is proved individually in an appropriate security model. The composability properties of this model then allow to transfer the security of the constituents to the overall construction.

2.2 Notations and terminology

We use the Bourbaki notation to denote sets of integers: for $a < b \in \mathbb{Z}$, $[\![a, b]\!] = \{a, a+1, \ldots, b\}$. For any set S, $|S|$ is its cardinality; if S is a set of integers and a is an integer, we denote by $S - a = \{s - a \mid s \in S\}$ the set S translated by $-a$.

Given a quantity x, we denote its shares by $x^{\mathbf{0}}$, …, $x^{\boldsymbol{n-1}}$; the set of these n shares is abbreviated as $x^{\star}$. For $0 \leqslant \boldsymbol{i} \leqslant \boldsymbol{j} < n$, $x^{\boldsymbol{i:j}} = (x^{\boldsymbol{i}}, x^{\boldsymbol{i+1}}, \ldots, x^{\boldsymbol{j}})$ is the subset of its shares having indexes from $\boldsymbol{i}$ to $\boldsymbol{j}$, inclusively.

We define the modulo *operator* as giving the remainder in the Euclidean division of two integers, that is, for $n \in \mathbb{Z}$ and $d > 1$, $m = n \bmod d$ is the unique integer such that $0 \leqslant m < d$ and $m \equiv n \pmod{d}$.

In this work, we will consider two classes of *masking schemes*, which determine the finite group over which the shares are considered. • We will talk of *b-bit Boolean masking* when the shares of a quantity x are elements of $\mathbb{F}_2^b$, recombined with bit-wise addition in $\mathbb{F}_2$: $x = x^{\mathbf{0}} \oplus \cdots \oplus x^{\boldsymbol{n-1}}$. For a set $S \subset \mathbb{F}_2^b$, we denote by $\mathcal{B}^n(S)$ the set of n-share Boolean sharings of elements of S. • We instead talk of *arithmetic masking modulo* m, for some arbitrary positive integer m, when the shares are considered over $\mathbb{Z}/(m\mathbb{Z})$, and are recombined with arithmetic addition modulo m: $x = (x^{\mathbf{0}} + \cdots + x^{\boldsymbol{n-1}}) \bmod m$. For a set $S \subset [\![0, m-1]\!]$, we denote by $\mathcal{A}_m^n(S)$ the set of n-share arithmetic sharings modulo m of elements of S.

2.3 Security model

We will prove the security of our construction in the Probe-Isolating Non-Interference (PINI) model of Cassiers and Standaert [10]. This model allows for gadgets that can be securely composed in the t-probing model [9, 19], in a way that is robust against glitches [14]. We recall the PINI definition as given in [9].

Definition 1 (Probe-Isolating Non-Interference [9]). *Given a gadget G, let I be a set of at most t_1 probes on its internal wires and O a set of probes on its output shares. Let A be the set of the share indexes of the shares in O, and $t_2 = |A|$. Let I and O be chosen such that $t_1 + t_2 \leqslant t$. The gadget G is t-PINI if and only if, for all I and O, there exist a set of at most t_1 share indexes B such that observations corresponding to I and O can be simulated using only the shares with indexes $A \cup B$ of each input sharing.*

We furthermore say that a gadget with n shares is PINI if it is $(n-1)$-PINI, in which case it is also t-PINI for any t [8]. PINI security is compatible with gadget composition, in the sense of the following property.

Proposition 1 (PINI composability [10]). *Any composition of t-PINI gadgets is itself t-PINI.*

A valuable aspect of PINI is that it is satisfied by share-isolating gadgets, that is, gadgets implemented by circuits which can be partitioned into sub-circuits each involving a single share index [10, Proposition 4] [8, Proposition 1].

About robust-probing security Provided the base gadgets are glitch-robust PINI, robustness against glitches is immediately transferred to the gadget composition [8]. Robustness against transition and against the combination of glitches and transitions has been studied by Cassiers and Standaert in [8].

2.4 ML-KEM ciphertext compression and integer rescaling

ML-KEM is a key encapsulation mechanism (KEM) over module lattices, recently standardized by NIST as FIPS 203 [15]. It is based on an underlying public-key encryption scheme (PKE), defining key generation, encryption and decryption. The PKE operates on polynomials having 256 coefficients over the field $\mathbb{F}_{q_\mathrm{K}}$, where $q_\mathrm{K} = 3329$ is a 12-bit prime. To secure the scheme against chosen-ciphertext attacks (CCAs), the PKE is converted into a KEM using a CCA transformation: in essence, the decapsulation operation consists in performing the PKE decryption, then immediately recomputing the PKE encryption, and checking that the recomputed ciphertext exactly matches with the received one.

To reduce the size of its ciphertexts, ML-KEM uses a coefficient-wise compression operation, which takes as input coefficients in $[\![0, q_\mathrm{K}-1]\!]$ and downscales their range to $[\![0, 2^d-1]\!]$, for some integer $0 < d < 12$. Each coefficient x undergoes a multiplication by $\frac{2^d}{q_\mathrm{K}}$, rounded to the nearest integer, as follows:

$$\mathrm{Compress}_d(x) = \left\lfloor \frac{2^d x}{q_\mathrm{K}} \right\rceil \bmod 2^d .$$

Our solution being not restricted to the specifics of ML-KEM compression, we will consider a more generic *integer rescaling* operation, defined by

$$\begin{aligned} \text{Rescale}_{q,q',r} : [\![0, q-1]\!] &\longrightarrow [\![0, q'-1]\!] \\ x &\longmapsto \left\lfloor \frac{xq'+r}{q} \right\rfloor \bmod q' , \end{aligned} \tag{1}$$

where q and q' are arbitrary integers greater than or equal to 2 (usually, $q' < q$), and r is an integer belonging to $[\![0, q-1]\!]$. Parameters q and q' of the rescaling respectively specify the input and output range, while r determines the rounding strategy: $r = 0$ results in truncation, while $r = \lfloor q/2 \rfloor$ results in rounding to the nearest neighbor (ties being rounded up). We can see that ML-KEM compression is a special case of this rescaling function: $\text{Compress}_d = \text{Rescale}_{q_\text{K}, 2^d, \lfloor q_\text{K}/2 \rfloor}$.

2.5 Previous works

The Compress operation is used in two contexts in ML-KEM. On one hand, during the decryption procedure, Compress_1 is used to extract message bits from a polynomial having coefficients following a relatively narrow distribution around 0 and $\lceil q_\text{K}/2 \rceil$. Applying one-bit compression to this polynomial gives out one bit of message for each input coefficient, with a value that depends on whether the input coefficient is closer to 0 or to $\lceil q_\text{K}/2 \rceil$. This specialization of the compression operation is also known as *decoding*. On the other hand, Compress_d is used at the end of the encryption procedure, to rescale each coefficient of the ciphertext to d bits, with $d \in \{4, 5, 10, 11\}$ depending on the security level and on which portion of the ciphertext is considered. These two contexts have been treated differently in the literature with regard to masking.

A solution to mask Compress_1 has been first proposed by Reparaz *et al.* [22], with a process that iteratively builds a lookup index from the two shares of the input coefficient, and derives from it the masked decoded value using a masked lookup table. Later, Oder *et al.* [21] proposed an entirely different technique: they convert the input shares to a power-of-two arithmetic sharing of the same value, apply some arithmetic transformations, followed by an arithmetic-to-Boolean conversion, and finally extract the sign bit of the obtained Boolean sharing.

Fritzmann *et al.* [16] simplify this process using explicit masking conversions, as well as secure addition over Boolean shares. They are also the first to propose a masked implementation of Compress_d for arbitrary d, as used during ML-KEM encryption. The authors start by noticing that when considering $\text{Compress}_d(x) = \lfloor x2^d/q_\text{K} \rceil \bmod 2^d$, the fractional part of $x2^d/q_\text{K}$ is never exactly $1/2$. The computation of the rounded division can thus tolerate a small error, bounded by $1/(2q_\text{K})$, and still return a correct value. Fritzmann *et al.* use this property to compute the operation over Boolean shares: they reformulate compression as multiplication by a higher-precision fixed-point approximation of $2^d/q_\text{K}$, followed by truncation of the unneeded precision. Coron *et al.* [12] generalize this approach to any masking order, as recalled in Algorithm 1, and explicitly compute the number of bits of additional precision, $\alpha = \lceil \log_2(nq_\text{K}) \rceil$.

Algorithm 1: Masked ciphertext compression from Coron *et al.* [12]

Parameter: Number of bits d of the compressed coefficient, number of extra bits of precision $\alpha = \lceil \log_2(nq_{\mathrm{K}}) \rceil$

Input: $x^\star \in \mathcal{A}_{q_{\mathrm{K}}}(\mathbb{F}_{q_{\mathrm{K}}})$

Output: $z^\star \in \mathcal{B}(\mathbb{F}_2^d)$ such that $\oplus_{\boldsymbol{i}} z^{\boldsymbol{i}} = \mathrm{Compress}_d(\sum_{\boldsymbol{i}} x^{\boldsymbol{i}} \bmod q_{\mathrm{K}})$

1 $y^\star = \lfloor 2^{d+\alpha} x^\star / q_{\mathrm{K}} \rceil \in \mathcal{A}_{2^{d+\alpha}}(\mathbb{Z}/2^{d+\alpha}\mathbb{Z})$ *// Fixed-point rescaling with α fraction bits*

2 $y^{\mathbf{0}} = y^{\mathbf{0}} + 2^{\alpha-1}$ *// Add $\frac{1}{2}$ for rounding*

3 $z^\star = \mathrm{A2B}_{2^{d+\alpha}}(y^\star)$ *// A2B conversion over $d+\alpha$ bits*

4 $z^\star = z^\star \gg \alpha$ *// Truncate the fractional part*

5 **return** $z^\star$

In some other works, the ciphertext compression is not masked at all, and a workaround is used for the security of the decapsulation, which crucially relies on the secrecy of the input to ciphertext compression. In this context, the goal is to compare the sensitive re-encrypted ciphertext with the public received ciphertext. Considering this, Bos *et al.* [6] choose not to compress the re-encrypted ciphertext, but to instead apply a special decompression to the received ciphertext, and perform the secure comparison using a masked interval check.

3 Masked integer rescaling

3.1 Principle

Let us now examine what happens when rescaling an integer represented over two arithmetic shares. Let us fix parameters q, q' and r. Consider a first-order arithmetic sharing modulo q, denoted by $(x^{\mathbf{0}}, x^{\mathbf{1}}) \in \mathcal{A}_q^2(\llbracket 0, q-1 \rrbracket)$. We recall that the value represented by this sharing is $x = (x^{\mathbf{0}} + x^{\mathbf{1}}) \bmod q$. We have

$$\mathrm{Rescale}_{q,q',r}(x) \equiv \left\lfloor \frac{xq'+r}{q} \right\rfloor \equiv \frac{xq'+r-(xq'+r) \bmod q}{q} \pmod{q'}.$$

Let us choose $r_0, r_1 \in \mathbb{Z}$ such that $r_0 + r_1 = r$. If we apply the rescaling on each share separately with rounding parameters r_0 and r_1 respectively, the error with respect to rescaling x directly is

$$\begin{aligned} \Delta^2_{q,q',r}(x^\star) &\equiv \mathrm{Rescale}_{q,q',r}(x) - \mathrm{Rescale}_{q,q',r_0}(x^{\mathbf{0}}) - \mathrm{Rescale}_{q,q',r_1}(x^{\mathbf{1}}) \pmod{q'}, \\ &\equiv \frac{(x^{\mathbf{0}}q'+r_0) \bmod q + (x^{\mathbf{1}}q'+r_1) \bmod q - (xq'+r) \bmod q}{q} \pmod{q'}. \end{aligned}$$

The numerator of this fraction belongs to $q\mathbb{Z} \cap \llbracket -q+1, 2q-2 \rrbracket = \{0, q\}$: it is indeed clearly equivalent to 0 modulo q since $x^{\mathbf{0}} + x^{\mathbf{1}} - x \equiv 0 \pmod{q}$ and $r_0 + r_1 = r$, and each of the three residues modulo q belongs to $\llbracket 0, q-1 \rrbracket$. Furthermore,

$$\begin{aligned} 0 &= ((x^{\mathbf{0}} + x^{\mathbf{1}})q' + (r_0 + r_1)) \bmod q \quad - \quad (xq'+r) \bmod q \\ &= \big((x^{\mathbf{0}}q' + r_0) \bmod q + (x^{\mathbf{1}}q' + r_1) \bmod q\big) \underbrace{\bmod q} \quad - \quad (xq'+r) \bmod q, \end{aligned}$$

where the modulo operator marked with the underbrace vanishes exactly when $(x^{\mathbf{0}}q' + r) \bmod q + (x^{\mathbf{1}}q' + r) \bmod q) < q$. The value of the error is thus

$$\Delta^2_{q,q',r}(x^\star) = \begin{cases} 1 & \text{if } (x^{\mathbf{0}}q' + r_0) \bmod q + (x^{\mathbf{1}}q' + r_1) \bmod q \geqslant q\,, \\ 0 & \text{otherwise}\,. \end{cases} \tag{2}$$

If we can compute this correction value Δ securely, integer rescaling can be performed over shared values by applying the rescaling share-wise with rounding parameters that sum to r, and adding the correction value to the rescaled shares.

3.2 Supporting gadgets

The secure implementation of our solution will rely on some supporting gadgets, some of which are already present in the literature.

Secure addition over Boolean shares Like related works (*e.g.*, [6,12,16]), our implementation of masked integer rescaling involves several masking schemes, and conversions between them. Practically, the input of masked integer rescaling will be an arithmetic sharing modulo q, its output will be an arithmetic sharing modulo q', and Boolean sharings will also be used internally. We will need to compute arithmetic addition over these, and consequently employ, as a central component, the masked gadget of *secure addition over Boolean shares*.

This gadget, introduced by Coron *et al.* [13], takes as inputs two values protected over Boolean shares, and gives as output a Boolean sharing of their sum. Several implementations of the gadget have been proposed, whether hardware-centric [2, 4, 16, 23] or software-centric [5, 13]. We denote by $\mathrm{SecAddBin}^n_w$ secure addition over n-share Boolean sharings of w bits.

In some cases, we will not use the sum, and only need to know whether an output carry is generated because the sum is larger than 2^w. We define for this purpose the secure-carry gadget $\mathrm{SecCarry}^n_w$, which can be straightforwardly obtained from $\mathrm{SecAddBin}^n_{w+1}$ by only keeping the most significant bit of its result. More efficient implementations are possible since only the carry bit is needed (as apparent from [4, fig. 3]), but they will not be detailed here.

As masked integer rescaling involves arithmetic sharings modulo an arbitrary integer q (a prime number in the case of ML-KEM compression), simple binary addition with wraparound is not sufficient: we will need to compute addition modulo q, and to additionally get information on whether the sum before modular reduction was already reduced or not. To this effect, we define the gadget of *secure modular addition with overflow indication*, denoted as SecAddOverflow. This gadget is easy to implement from the literature, which gives several constructions for the gadget of secure addition modulo q, SecAdd_q [2, 5, 16]. As all these implicitly determine whether the sum must be reduced by subtracting the modulus from it, outputting this information beside the sum gives SecAddOverflow. As an example, we show in Algorithm 2 how to apply this transformation to the SecAdd_q of Fritzmann *et al.* [16].

We define a specialization of SecAddOverflow, which discards the sum and only outputs the modular-overflow indication: SecOverflow. As seen in Algorithm 2, this removes the need for the secure-multiplexing (SecMux) gadget.

Algorithm 2: SecAddOverflow: Secure modular addition with overflow, derived from [16, alg. 7]

Parameter: Modulus q, sharing width $w = \lceil \log_2(q) \rceil$
Input: Summands $x^\star, y^\star \in \mathcal{B}_w^n([\![0, q-1]\!])$
Output: Modular sum $z^\star \in \mathcal{B}_w^n([\![0, q-1]\!])$ such that $z = (x + y) \bmod q$,
modular-overflow indication $c^\star \in \mathcal{B}(\{0,1\})$, such that $b = 1 \Leftrightarrow x + y \geqslant q$

1 $s^\star = \text{SecAddBin}_{w+1}^n(x^\star, y^\star)$
2 $s'^\star = \text{SecAddBin}_{w+1}^n(s^\star, (2^{w+1} - q, 0))$
3 $c^\star = s'^\star \gg w$
4 $z^\star = \text{SecMux}_w^n(s^\star, s'^\star, c^\star)$ *// Securely multiplex between $s^\star$ and $s'^\star$ based on $c^\star$*
5 **return** $z^\star, c^\star$

Note that, while the q parameter of $\text{SecAddOverflow}_q^n$ and SecOverflow_q^n indicates the modulus, the w parameter of SecAddBin_w^n and SecCarry_w^n denotes the number of bits of the operands, so that the corresponding modulus is 2^w. In the rest of this work, we assume all these gadgets to satisfy PINI security[4].

Single-bit Boolean to arithmetic conversion We will make use a single-bit Boolean-to-Arithmetic conversion satisfying PINI security. This gadget, that we denote by B2Abit_q^n, gets as input a single bit expressed over n Boolean shares, and converts it to arithmetic shares modulo q, with no restriction on the value of the modulus q. Schneider *et al.* [24] proposed such a gadget, with lower complexity than a conversion from Boolean to modulo-q arithmetic masking supporting any input between 0 and $q-1$. They prove that their gadget satisfies SNI (Strong Non-Interference) security, but it happens that for the special case of gadgets having a single input sharing and a single output sharing, SNI implies PINI, giving us the result of Lemma 1.

Lemma 1. *The* B2Abit_q^n *gadget is PINI.*

Proof. Cassiers and Standaert introduce in [10] the notion of t-MIMO-SNI, which they show to imply t-PINI [10, Proposition 6]. It happens that for gadgets having a single input sharing and a single output sharing, t-MIMO-SNI is exactly

[4] For example, the HPC2 SecAddBin of Bache *et al.* [4] can be employed together with a SecMux gadget built from HPC2 AND gates [9], in the composition detailed in Algorithm 2 for the modular operations. Or, the proposal of Assael *et al.* [2], which satisfies the stronger O-PINI security notion, can be used to directly provide the power-of-two and the modular sums, assuming its internal carry and modular-overflow indication are made available externally. The PINI security of both choices is easily checked thanks to the permissive properties of PINI regarding composition.

equivalent to t-SNI. t-SNI gadgets with a single input sharing and a single output sharing are thus also t-PINI. Since Schneider *et al.* proved that their $\mathrm{SecB2A}_{q\text{-}Bit}$ in [24, Algorithm 5] is t-SNI, the conclusion follows. □

Fractional rescaling In order to compute the rescaling efficiently, we define an extension of integer rescaling, that not only computes the Rescale function, but also the error introduced by it through rounding. We define this operation by

$$\begin{aligned}\mathrm{DivMod}_{q,q',r} : [\![0, q-1]\!] &\longrightarrow [\![0, q'-1]\!] \times [\![0, q-1]\!] \\ x &\longmapsto \left(\left\lfloor \frac{xq'+r}{q} \right\rfloor \bmod q', (xq'+r) \bmod q \right). \quad (3)\end{aligned}$$

A very important aspect of this function is that it computes a single Euclidean division, $(xq'+r) \div q$, and employs both its quotient (further reduced modulo q') and its remainder. In general, computing both outputs of the Euclidean division is only marginally more costly than computing either.

We call the first output of DivMod its *integral* result, which corresponds to Rescale, and the second output its *fractional* result, because it represents the rescaling error in units of $1/q$. Note that DivMod satisfies the following property:

$$(y, f) = \mathrm{DivMod}_{q,q',r}(x) \quad \Longrightarrow \quad qy + f \equiv xq' + r \pmod{qq'}. \quad (4)$$

SplitShares We finally define a gadget that, despite being essentially empty, will be crucial to our security proof. Given $m, n, w \geqslant 1$, we define the gadget $\mathrm{SplitShares}_w^{m,n}$ that takes two w-bit input sharings with m and n shares respectively, and constructs from them two sharings each having $m+n$ shares by adjoining null shares to the initial sharings. The definition of the gadget is given in Algorithm 3, and illustrated in Fig. 1. Since the definitions that we use need gadgets to have the same number of shares in all of their input and output sharings, we regard the two input sharings of SplitShares as a single one of order $m+n$. We highlight that the indexes that correspond to the null shares are essential to the PINI security of the gadget, since they ensure that the input shares keep the same index at the output. This property allows for the gadget to satisfy PINI security even though it does not include any refresh.

The purpose of SplitShares is to allow *two*-input gadgets having $m+n$ shares to be fed a *single* sharing of order $m+n$, by splitting it into two sharings of orders m and n respectively, to securely transform the input sharing while respecting its order. The intuition behind the validity of the approach is the following: since PINI gadgets emulate share-isolating ones, we can freely move the shares from one input sharing to another, mix them with null shares, and compute non-linear PINI gadgets over the rearranged sharings, as long as the share indexes are preserved by the rearrangement.

Proposition 2. *The* SplitShares *gadget is PINI.*

Proof. SplitShares being share-isolating, the result follows from [10, Prop. 4]. □

Algorithm 3: $\mathrm{SplitShares}_w^{m,n}$ gadget

Input: Sharing $(a^{\mathbf{0}}, \ldots, a^{\boldsymbol{m+n-1}}) \in (\mathbb{F}_2^w)^{m+n}$ (arbitrary masking scheme)
Output: Sharings $(x^{\mathbf{0}}, \ldots, x^{\boldsymbol{m+n-1}}) \in (\mathbb{F}_2^w)^{m+n}$, $(y^{\mathbf{0}}, \ldots, y^{\boldsymbol{m+n-1}}) \in (\mathbb{F}_2^w)^{m+n}$

for $\boldsymbol{i} = 0$ to $m+n-1$ **do**
  $x^{\boldsymbol{i}} = a^{\boldsymbol{i}}$ **if** $\boldsymbol{i} < m$ **else** 0
  $y^{\boldsymbol{i}} = 0$ **if** $\boldsymbol{i} < m$ **else** $a^{\boldsymbol{i}}$
end

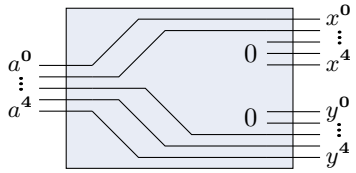

Fig. 1. Diagram of the $\mathrm{SplitShares}_w^{2,3}$ gadget

Algorithm 4: $\mathrm{SecDivMod}_{q,q',r}^n$ gadget: share-wise fractional rescaling

Parameter: Input and output moduli q, q', number of shares $n \geqslant 2$, rounding parameters $r_0, \ldots, r_{n-1}$
Input: Arithmetic sharing modulo q, $(x^{\mathbf{0}}, \ldots, x^{\boldsymbol{n-1}}) \in \mathcal{A}_q^n(\llbracket 0, q-1 \rrbracket)$
Output: Arithmetic sharings mod q' and mod q: $(y^{\mathbf{0}}, \ldots, y^{\boldsymbol{n-1}}) \in \mathcal{A}_{q'}^n(\llbracket 0, q'-1 \rrbracket)$ and $(f^{\mathbf{0}}, \ldots, f^{\boldsymbol{n-1}}) \in \mathcal{A}_q^n(\llbracket 0, q-1 \rrbracket)$, s.t. $\forall \boldsymbol{i}, (y^{\boldsymbol{i}}, f^{\boldsymbol{i}}) = \mathrm{DivMod}_{q,q',r_{\boldsymbol{i}}}(x^{\boldsymbol{i}})$

for $\boldsymbol{i} = 0$ to $n-1$ **do** $(y^{\boldsymbol{i}}, f^{\boldsymbol{i}}) = \mathrm{DivMod}_{q,q',r_{\boldsymbol{i}}}(x^{\boldsymbol{i}})$
return $y^\star, f^\star$

We also define the $\mathrm{SecDivMod}_{q,q',r}^n$ gadget as the share-wise application of the $\mathrm{DivMod}_{q,q',r}$ function over an arithmetic sharing, according to Algorithm 4.

Proposition 3. *The* SecDivMod *gadget is PINI*

Proof. SecDivMod is share-isolating, from which the result follows. □

3.3 Secure implementation at the first order

We show in Fig. 2 the composition of gadgets through which we implement first-order masked rescaling from q to q', with rounding parameter r. In this diagram, each wire carries a first-order sharing, either in the Boolean domain (blue wires) or in the arithmetic domain (brown wires). We use three share-isolating gadgets, represented in light blue: $\mathrm{SecDivMod}_{q,q',r}^2$, $\mathrm{SplitShares}_w^{1,1}$, and addition modulo q'. Two nonlinear gadgets, $\mathrm{SecOverflow}_q^2$ and $\mathrm{B2A}_{q'}$ are also needed, and are represented in orange. The arithmetic shares modulo q of the input value $x^\star$ are first subjected to fractional rescaling through SecDivMod. The integral result

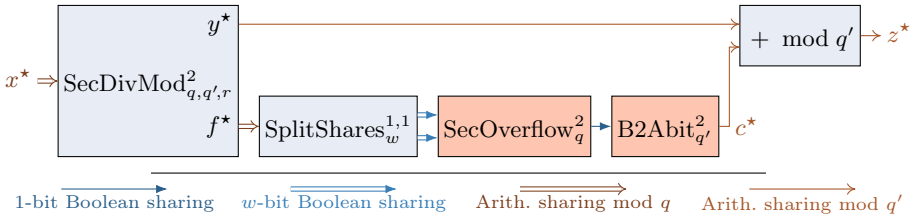

Fig. 2. First-order masked computation of integer rescaling. Symbol w denotes $\lceil \log_2 q \rceil$. Each wire carries a 1st-order sharing among the types listed at the bottom.

$y^\star$ is represented over mod-q' arithmetic shares, while the fractional result is still on mod-q arithmetic shares. This latter result is further split into a pair of Boolean shares (each having $w = \lceil \log_2 q \rceil$ bits), which are added together to check whether their sum is greater than q, thereby implementing the Δ^2_q function. This indication, output by SecOverflow over a single-bit Boolean sharing, is converted to an arithmetic sharing modulo q': $c^\star$. Finally, the integral result $y^\star$ of fractional rescaling is summed share-wise with the correction value $c^\star$, giving as $z^\star$ the expected result of rescaling, over arithmetic shares modulo q'.

The whole process is also formalized in Algorithm 5, with a proposed variant which differs in how the fractional correction is computed based on the available gadgets. In the first solution, as in Fig. 2, we use SecOverflow^2_q to perform the comparison with q directly (Lines 1.2 to 1.3), while in the second solution, we subtract q from $f^{\mathbf{0}}$ beforehand, and simply check whether summing this result with $f^{\mathbf{1}}$ generates an output carry (Lines 2.2 to 2.3). As this first-order gadget is a special case of the high-order gadget described next, we do not yet prove its security, and will prove it generically for any order.

Algorithm 5: SecRescale$^2_{q,q',r}$ gadget: first-order secure integer rescaling

Parameter: Input and output moduli q and q', input-modulus width $w = \lceil \log_2 q \rceil$, rounding parameters r_0, r_1 such that $r_0 + r_1 = r$

Input: Arithmetic sharing modulo q, $(x^{\mathbf{0}}, x^{\mathbf{1}}) \in \mathcal{A}^2_q([\![0, q-1]\!])$

Output: Arith. sharing mod. q', $(z^{\mathbf{0}}, z^{\mathbf{1}}) \in \mathcal{A}^2_{q'}([\![0, q'-1]\!])$ s.t. $z = \text{Rescale}_{q,q',r}(x)$

Solution 1 (as in Fig. 2)	**Solution 2** – underlines show differences
1.1 $y^\star, f^\star = \text{SecDivMod}^2_{q,q',r}(x^\star)$	2.1 $y^\star, f^\star = \text{SecDivMod}^2_{q,q',r}(x^\star)$
// Boolean sharings of $f^{\mathbf{0}}$ and of $f^{\mathbf{1}}$	*// Bool. sharings of $f^{\mathbf{0}}+2^w-q$ and of $f^{\mathbf{1}}$*
1.2 $g^\star_0, g^\star_1 = \text{SplitShares}^{1,1}_w(f^\star)$	2.2 $g^\star_0, g^\star_1 = \text{SplitShares}^{1,1}_w((f^{\mathbf{0}} \underline{+ 2^w - q}, f^{\mathbf{1}}))$
1.3 $b^\star = \underline{\text{SecOverflow}}^2_q(g^\star_0, g^\star_1)$	2.3 $b^\star = \underline{\text{SecCarry}}^2_w(g^\star_0, g^\star_1)$
1.4 $c^\star = \text{B2Abit}^2_{q'}(b^\star)$	2.4 $c^\star = \text{B2Abit}^2_{q'}(b^\star)$ . . . . *// 1-bit* B2A
1.5 $z^\star = (y^\star + c^\star) \bmod q'$	2.5 $z^\star = (y^\star + c^\star) \bmod q'$ *// Share-wise add.*
1.6 **return** $z^\star$	2.6 **return** $z^\star$

3.4 Constructing high-order gadgets by share concatenation of low-order gadgets

We establish here a result that will be helpful in proving the high-order security of our construction: given two PINI gadgets g', g'' having n', n'' shares respectively, we can build from them a PINI gadget G with $n'+n''$ shares by mapping the n' first (respectively n'' last) share indexes of the input and output sharings of G to the input and output sharings of g' (respectively g''). We call gadget G the *share concatenation* of g' and g''. We start by formally defining this concept.

Definition 2 (Share concatenation of gadgets). *Let g', g'' be two distinct gadgets each having u input sharings and v output sharings, such that g' satisfies:*

- *g' has n' shares,*
- *g' has inputs $({U'}_j^{\boldsymbol{\delta}})_{0\leqslant j<u, 0\leqslant \boldsymbol{\delta}<n'}$, where ${U'}_j^{\star}$ is its j^{th} input sharing,*
- *g' has outputs $({V'}_j^{\boldsymbol{\delta}})_{0\leqslant j<v, 0\leqslant \boldsymbol{\delta}<n'}$, where ${V'}_j^{\star}$ is its j^{th} output sharing,*

and g'' satisfies equivalent properties, replacing all primes $'$ with double primes $''$.

Denote by G the gadget with $n'+n''$ shares, u input sharings and v output sharings, constructed from the union of g' and g'' in the following way:

- *for $0 \leqslant j < u$, the j^{th} input sharing of G is $(U_j^{\boldsymbol{\delta}})_{0\leqslant\delta<n'+n''}$, where $U_j^{\boldsymbol{\delta}} = {U'}_j^{\boldsymbol{\delta}}$ if $\delta < n'$ and $U_j^{\boldsymbol{\delta}} = {U''}_j^{\boldsymbol{\delta}-n'}$ otherwise,*
- *for $0 \leqslant j < v$, the j^{th} output sharing of G is $(V_j^{\boldsymbol{\delta}})_{0\leqslant\delta<n'+n''}$, where $V_j^{\boldsymbol{\delta}} = {V'}_j^{\boldsymbol{\delta}}$ if $\delta < n'$ and $V_j^{\boldsymbol{\delta}} = {V''}_j^{\boldsymbol{\delta}-n'}$ otherwise.*

We call G the share concatenation *of g' and g'', and inductively extend this notion to the share concatenation of more than two gadgets.*

Proposition 4. *If G is the share concatenation of two gadgets g' and g'', both of which are PINI, then G is likewise PINI. The same holds for the share concatenation of more than two gadgets.*

Proof. Consider g', g'', G satisfying the hypotheses of Definition 2. Let $t = n' + n'' - 1$. Let I and O be sets of internal and output probes on G, with $|I| = t_1$, and when denoting by A the set of the share indexes in O, $|A| = t_2$, with $t_1 + t_2 = t$.

We partition I into $I' \cup I''$ such that I' is the set of internal probes belonging to g' and I'' is the set of internal probes belonging to g''. We also partition $O = O' \cup O''$ in the same way. Let us denote by $A' \subset [\![0, n'-1]\!]$ the set of share indexes in O', and by $A'' \subset [\![n', n'+n''-1]\!]$ the set of share indexes in O''.

Since g' is t'-PINI for any t', there exists a set $B' \subset [\![0, n'-1]\!]$ of at most $|I'|$ share indexes such that observations corresponding to I' and O' can be simulated using only the shares with indexes $A' \cup B'$ of each input sharing of g'.

Similarly, since g'' is t''-PINI for any t'', there exists a set $B'' \subset [\![n', n'+n''-1]\!]$ of at most $|I''|$ share indexes such that observations corresponding to I'' and O'' can be simulated using only the shares with indexes $(A'' - n') \cup (B'' - n')$ of each input sharing of g'', where the translation remaps the share indexes to their original positions in g''.

We define $B = B' \cup B''$. We trivially have $|B| = |B'| + |B''| \leqslant |I'| + |I''| = t_1$. We can separately simulate the probes in I' and O' on one hand, in I'' and O'' on

the other hand, using the two above-mentioned simulators, as we know the input shares of g' with indexes in $A' \cup B'$ and the input shares of g'' with indexes in $(A''-n') \cup (B''-n')$. As G has $n = n'+n''$ shares and is $(n-1)$-PINI, it is PINI. □

Note that the proposition requires that g' and g'' be secure at their maximum security order given their number of shares: orders $n'-1$ and $n''-1$ respectively, so that they are t-PINI for any t. This proposition can be seen as an extension of [8, Proposition 1], which states that share-isolating gadgets are (O-)PINI. We also note that a related property of PINI, *gadget embedding*, was proved by Bronchain and Cassiers [7]; we do not use this latter property here since it tends to complicate the reasoning about the actual order of sharings.

Extending this result to robust-probing security The above result can be extended to the O-PINI security notion of Cassiers and Standaert [8]: *any share concatenation of O-PINI gadgets is O-PINI.* We recall that this stronger security notion additionally requires simulating, in the same context as above, the output shares having index in B. The proof can be adapted to this context by noticing that the output shares of G with indexes in B exactly correspond to the output shares of g' with indexes in B' together with the output shares of g'' with indexes in $B'' - n'$, which can be simulated under the assumption that g' and g'' are O-PINI. The result also holds for robust probing, provided the probe expansion scheme is compatible with the partitioning of G into g' and g'', *i.e.* expanding probes in G is equivalent to expanding them in g' and in g'' separately. In particular, it holds for glitch+transition-robust probing if the members of the share concatenation are implemented by disjoint structural gates and wires. We refer the reader to [8] for the definition of these terms.

3.5 High-order masked integer rescaling

The general principle used for first-order masked integer rescaling, namely associating a share-wise fractional rescaling with a nonlinear correction step, can actually be applied at any order. However, the computation of the correction becomes more complex, because the amplitude of the accumulated rounding error is proportional to the masking order: with n shares, this error is comprised between 0 and $n-1$.

Using the same notation as before, the correction function for n shares becomes

$$\Delta^n_{q,q',r}(x^{\mathbf{0}}, \ldots, x^{\mathbf{n-1}}) = \left\lfloor \frac{\sum_{\boldsymbol{i}=0}^{n-1}((x^{\boldsymbol{i}}q' + r_{\boldsymbol{i}}) \bmod q)}{q} \right\rfloor \in [\![0, n-1]\!], \qquad (5)$$

which corresponds to summing the fractional results of share-wise rescaling, dividing the result by q, and only keeping the integral part of the quotient. While this truncated division could be performed by comparing with q in the first-order case, this is no longer possible at higher orders. Our solution, instead, is to add the shares progressively, checking each time if a reduction modulo q is needed, and to count how many reductions have been performed.

Algorithm 6: $\mathrm{SecRescale}^n_{q,q',r}$ gadget: high-order secure integer rescaling

Parameter: Input and output moduli q, q'; width $w = \lceil \log_2 q \rceil$ of input modulus; number of shares $n \geqslant 2$; rounding param. $r_0, \ldots, r_{n-1}$ with $\sum_i r_i = r$

Input: Arithmetic sharing modulo q: $x^\star \in \mathcal{A}^n_q(\llbracket 0, q-1 \rrbracket)$

Output: Arithmetic sharing mod. q': $z^\star \in \mathcal{A}^n_{q'}(\llbracket 0, q'-1 \rrbracket)$, s.t. $z = \mathrm{Rescale}_{q,q',r}(x)$

1 $y^\star, f^\star = \mathrm{SecDivMod}^n_{q,q',r}(x^\star)$

2 $J = (1)_{0 \leqslant i < n}$ . . . . . *// $J = (J_i)_{0 \leqslant i < |J|}$ is a partition of the number of shares*

3 **while** $|J| > 1$ **do** . . *// Loop until J has been reduced to the trivial partition (n)*

4 $\quad \boldsymbol{j} = 0$

5 $\quad$ **for** $i = 0$ to $\lfloor |J|/2 \rfloor - 1$ **do**

6 $\quad\quad \boldsymbol{j'} = \boldsymbol{j} + J_{2i} + J_{2i+1} - 1$

7 $\quad\quad F_0^{\boldsymbol{j}:\boldsymbol{j'}}, F_1^{\boldsymbol{j}:\boldsymbol{j'}} = \mathrm{SplitShares}_w^{J_{2i}, J_{2i+1}}(f^{\boldsymbol{j}:\boldsymbol{j'}})$

8 $\quad\quad f^{\boldsymbol{j}:\boldsymbol{j'}}, b^{\boldsymbol{j}:\boldsymbol{j'}} = \mathrm{SecAddOverflow}_q^{J_{2i}+J_{2i+1}}(F_0^{\boldsymbol{j}:\boldsymbol{j'}}, F_1^{\boldsymbol{j}:\boldsymbol{j'}})$ *// Sum sharings pair-wise*

9 $\quad\quad c^{\boldsymbol{j}:\boldsymbol{j'}} = \mathrm{B2Abit}_{q'}^{J_{2i}+J_{2i+1}}(b^{\boldsymbol{j}:\boldsymbol{j'}})$ . . . . . *// One-bit* B2A *conversion*

10 $\quad\quad \boldsymbol{j} = \boldsymbol{j'} + 1$

11 $\quad$ **end**

12 $\quad$ **if** $|J| \equiv 1 \pmod 2$ **then**

13 $\quad\quad \boldsymbol{j'} = n - 1$

14 $\quad\quad c^{\boldsymbol{j}:\boldsymbol{j'}} = 0, \ldots, 0$

15 $\quad$ **end**

16 $\quad y^\star = (y^\star + c^\star) \bmod q'$ *// Share-wise modular add. of current correction value*

17 $\quad J = (J_{2i} + J_{2i+1})_{0 \leqslant i < \lceil |J|/2 \rceil}$ . . . . . *// Merge pairs of terms of J; $J_{|J|} = 0$*

18 **end**

19 $z^\star = y^\star$

20 **return** $z^\star$

We describe in Algorithm 6 the implementation of secure integer rescaling with n shares. As before, we start by applying fractional rescaling share-wise. This operation gives two sharings: $y^\star$, and $f^\star$. The former is an approximation of the rescaled result, which may underestimate it by up to $n-1$ units, while the latter contains the rescaling error. The key part of our algorithm, contained in Lines 2 to 18, consists in progressively migrating this error from $f^\star$ to $y^\star$. A graphical representation of this process is also given in Fig. 3 for seven shares.

We compute the correction by summing the shares $f^\star$ modulo q in a binary tree structure, and adding 1 to y every time a modular sum overflows from $\llbracket 0, q-1 \rrbracket$. The sum of the fractional parts is protected through Boolean masking, with a number of shares progressively increased to match the number of fractional shares involved in each partial sum[5]. In Algorithm 6, the progress of

[5] This kind of recursive implementation over increasing numbers of shares has been previously used, in particular, by Coron *et al.* [13].

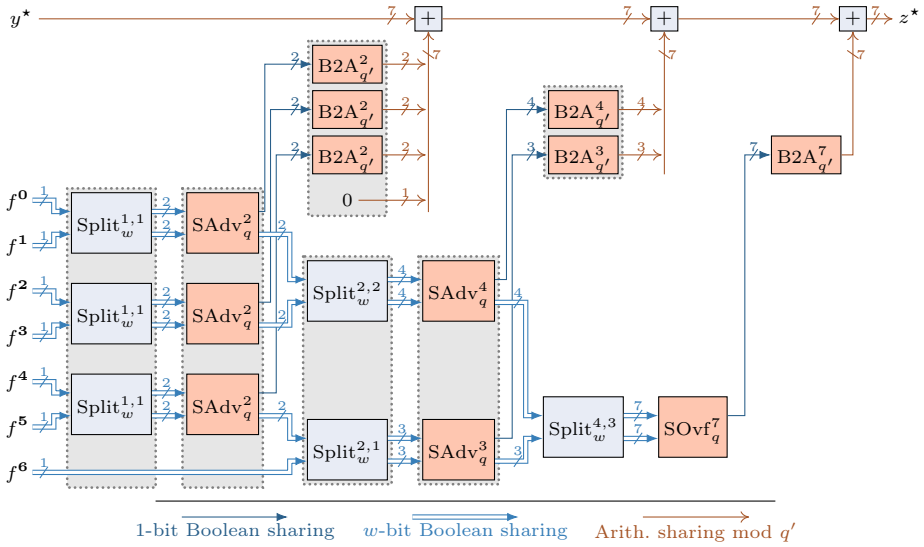

Fig. 3. Correction of rescaling from q to q' with seven shares. We use $w = \lceil \log_2 q \rceil$. Each wire carries a sharing among the types listed at the bottom, with the number of shares indicated next to every wire. Gadgets grouped within gray regions constitute 7-share gadgets through share concatenation. Abbreviations Split, SAdv, SOvf, and B2A stand for SplitShares, SecAddOverflow, SecOverflow, and B2Abit respectively. The top and bottom output sharings of SAdv respectively correspond to its overflow and sum outputs.

the summation is represented by J, a composition of the number of shares n, that is, an ordered list of positive integers that sum to n. Throughout the algorithm, each term of J indicates how many fractional shares have been summed together, and the number of shares of the corresponding sum.

Practically, each fractional share $f^{\boldsymbol{i}}$ is initially a 0^{th} order (one-share) sharing: this is expressed by J being the sequence of n ones. Each iteration of the loop starting at Line 3 applies one level of the summation tree. The first iteration of the inner loop at Line 5 takes the first pair of Boolean sharings from $f^\star$, with J_0 and J_1 shares, and replaces them with a Boolean sharing of their sum modulo q, this time over $J_0 + J_1$ shares. The masked bit b indicating whether modular reduction occurred, represented over $J_0 + J_1$ shares as well, is converted into a modulo-q' arithmetic sharing, c. The following iterations proceed likewise on the rest of $f^\star$ and $c^\star$, merging every pair of Boolean sharings into a sharing of their sum while preserving the overall number of shares. If composition J had an odd number of terms at this level of the tree (Line 12), the lone last sharing is kept unchanged in $f^\star$, and the corresponding shares of $c^\star$ are set to zero[6].

[6] When the number of shares is not a power of two, different shapes are possible for the correction tree. The optimal tree will depend on how exactly the runtime of the base gadgets evolves with their masking order, and may not be the one we choose here.

At this point, $c^\star$ contains a modular arithmetic sharing of the number of modular reduction that have occurred at this level of the summation tree: indeed, notice that concatenating arithmetic sharings gives a higher-order sharing of their sum. $c^\star$ can then be summed share-wise into $y^\star$ to apply the corresponding correction. Finally, the composition J of the number of shares is updated by summing its terms in pairs (leaving the last lone term unchanged if applicable) to reflect how the pairs of sharings have been merged. The next iteration of the outer loop then computes the following level of the summation tree, until J has been reduced to a single term. This condition indeed means that all fractional shares have been summed, and all required corrections have been applied to $y^\star$.

Comparing Algorithm 6 with Fig. 3, one can notice a difference that is small but sometimes important for performance. On one hand, the algorithm involves the same gadgets SplitShares, SecAddOverflow and B2A at each level of the tree for clarity. On the other hand, Fig. 3 uses SecOverflow instead of SecAddOverflow at the last level of the tree: only the modular-overflow indication is needed, and the sum can be discarded. Depending on the implementation of SecAddOverflow and SecOverflow, this optimization can bring a significant performance gain since this gadget is masked at the highest order, n.

We have not discussed yet how to choose the set of rounding parameters, $r_0, \ldots, r_{n-1}$, aside from the constraint that their sum is r. In fact, this choice does not matter security-wise, and has negligible influence performance-wise, apart from saving an arithmetic addition and a modular reduction by q' within DivMod if the corresponding rounding parameter is zero. We thus suggest two strategies to distribute the rounding parameter.

- If the number of shares n divides r, all rounding parameters can be equal: $\forall i \in [\![0, n-1]\!], r_i = r/n$. This is notably the case for ML-KEM ($r = \lfloor q_\mathrm{K}/2 \rfloor = 1664$) when the number of shares is a power of two, up to $n = 2^7$.
- In other cases, the simplest choice is to set $r_0 = r$ and to null the other rounding parameters, $\forall i \in [\![1, n-1]\!], r_i = 0$.

4 Analysis and comparison with the literature

4.1 Correctness and security

To prove the correctness and security of Algorithm 6, we start by analyzing the properties of the outer loop. We first state in Lemma 2 an invariant satisfied by J. The proofs are in Appendix A.

Lemma 2 (First invariant of the loop at Line 3 of Algorithm 6). *J is a composition of n at all times, that is, $\sum_i J_i = n$, and for all i, $J_i \geqslant 1$.*

For the following, we introduce an auxiliary function, HybridUnmask. This function takes two tuples (finite ordered sets) J and f of positive integers such that $\sum_i J_i = |f|$, and is recursively defined for the empty tuples by

$$\mathrm{HybridUnmask}(\varnothing, \varnothing) = 0\,,$$

and for nonempty J and f,

$$\text{HybridUnmask}(J, f) = \left(\bigoplus_{i=0}^{J_0-1} f_i\right) + \text{HybridUnmask}((J_1, ..., J_{|J|-1}), (f_{J_0}, ..., f_{|f|}))\,.$$

This function thus splits f in groups of size given by the elements of J, reduces each group to a single element through a bitwise XOR, and sums over all groups.

Lemma 3. *The computation in Lines 7 to 9 of Algorithm 6 satisfies*

$$\bigoplus_{\boldsymbol{k}=\boldsymbol{j}}^{\boldsymbol{j'}} f^{\boldsymbol{k}} + q \sum_{\boldsymbol{k}=\boldsymbol{j}}^{\boldsymbol{j'}} c^{\boldsymbol{k}} \equiv \text{HybridUnmask}((J_{before,2i}, J_{before,2i+1}), f_{before}^{\boldsymbol{j}:\boldsymbol{j'}}) \pmod{qq'}\,,$$

where symbols subscripted with $_{before}$ refer to the value of the corresponding variables before the execution of said lines, and symbols without this subscript refer to their value after execution.

Lemma 4 (Second invariant of the loop at Line 3 of Algorithm 6). *Considered at Line 3, the following equivalence holds across all iterations of the outer loop:*

$$\text{HybridUnmask}(J, f^\star) + q \sum_{\boldsymbol{i}=0}^{n-1} y^{\boldsymbol{i}} \equiv \sum_{\boldsymbol{i}=0}^{n-1} (x^{\boldsymbol{i}} q' + r_{\boldsymbol{i}}) \pmod{qq'}\,.$$

We can now prove that the proposed algorithm terminates and that it correctly computes the Rescale function (Theorem 1).

Theorem 1 (Correctness and termination of Algorithm 6). *Algorithm 6 terminates after $\lceil \log_2 n \rceil$ iterations of the loop at Line 3. It correctly computes $z^\star$ such that $(\sum_{\boldsymbol{i}=0}^{n-1} z^{\boldsymbol{i}}) \bmod q' = \text{Rescale}_{q,q',r}((\sum_{\boldsymbol{i}=0}^{n-1} x^{\boldsymbol{i}}) \bmod q)$.*

We finally prove the security of our construction (Theorem 2) through a compositional approach: low-order gadgets are merged into n-share gadgets through share concatenation, and the resulting gadgets are composed under our working assumption of PINI base gadgets.

Theorem 2 (Security of Algorithm 6). *Algorithm 6 implements a PINI gadget over n shares.*

Proof. Consider an arbitrary iteration of the outer loop of Algorithm 6, and iteration i of the inner loop starting at Line 5. $\text{SplitShares}_w^{J_{2i}, J_{2i+1}}$ is a PINI gadget with $J_{2i} + J_{2i+1}$ shares by Proposition 2. By assumption, the same holds for SecAddOverflow and B2Abit (see Section 3.2 and Lemma 1). Through the composition of PINI gadgets, each iteration of the inner loop is thus itself a PINI gadget over $J_{2i} + J_{2i+1}$ shares. Furthermore, if $|J|$ is odd, Lines 12 to 15

implement a gadget over $J_{|J|-1}$ shares, that ignores its $f^{\boldsymbol{j}:\boldsymbol{j}'}$ input sharing and outputs an all-zero sharing as $c^{\boldsymbol{j}:\boldsymbol{j}'}$. Being share-isolating, it is similarly PINI.

Therefore, Lines 5 to 12, in which the inner loop has been unrolled, implement an n-share share concatenation of PINI gadgets, which is itself PINI by Proposition 4.

The share-wise addition at Line 16 and the $\mathrm{SecDivMod}^n_{q,q',r}$ gadget at Line 1 satisfy the same security property (by linearity and by Proposition 3). The whole algorithm is thus a composition of n-share PINI gadgets. □

4.2 Optimization of the upper level of the correction tree

For simplicity, our description of the gadget in Algorithm 6 uses a homogeneous correction tree, having the same types of gadgets at each level. However, as already shown in Fig. 3, we can actually simplify the upper level of the tree by replacing $\mathrm{SecAddOverflow}_q$ with $\mathrm{SecOverflow}_q$ since only the overflow output of the former gadget is needed.

Furthermore, $\mathrm{SecOverflow}_q$ itself can be replaced with a composition of gadgets of possibly lower complexity (depending on their exact implementation). We recall that Fritzmann *et al.* [16] proposed, in the context of computing the secure addition modulo q of two quantities a and b, to determine whether modular overflow occurs by instead calculating $a + (b - q)$, and checking the sign of the result, which is directly obtained by checking the output carry of the sum.

We can use the same technique in the upper level of the reduction tree: instead of checking whether the last sum of fractional shares is greater than q, we securely add $2^w - q$ (we recall that $w = \lceil \log_2 q \rceil$) to one of the fractional shares before summing them, and simply check with SecCarry whether the subsequent sum has a nonzero output carry, as represented in Fig. 4. This transformation is advantageous, since the addition of $2^w - q$ now needs a lower-order protection: still referring to Fig. 4, computing this operation over three shares is sufficient for security. In contrast, the $\mathrm{SecOverflow}_q$ gadget had to compute it internally over seven shares.

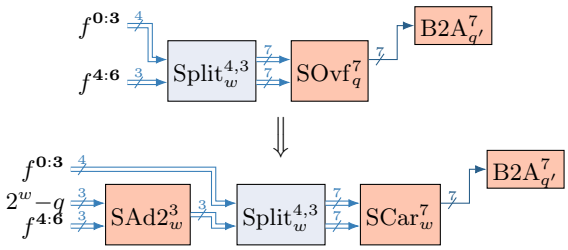

Fig. 4. Optimization of the upper level of the masked correction tree. The upper diagram, extracted from Fig. 3, is optimized into the lower one. Abbreviations Split, SOvf, SCar and SAd2 stand for SplitShares, SecOverflow, SecCarry and SecAddBin respectively.

4.3 Comparison with previous works

In this subsection, we compare our solution based on secure integer rescaling with state-of-the art proposals for the masking of ML-KEM compression. We thus assume that $q' = 2^d$ for some integer $d \leqslant 11$. We start by analyzing the running time of our construction. For simplicity, we focus on a power-of-two number of shares, $n = 2^N$, so that the correction is applied along a well-balanced binary tree. If we optimize the upper level of the tree as discussed in Section 4.2, and neglect the small running time of the linear gadgets (in particular SecDivMod), SecRescale has a time complexity of

$$T\big(\mathrm{SecRescale}^{2^N}_{q,q',r}\big) \approx T\big(\mathrm{SecAddBin}^{2^{N-1}}_{w}\big) + T\big(\mathrm{SecCarry}^{2^N}_{w}\big) + \sum_{i=1}^{N-1} 2^{N-i} T\big(\mathrm{SecAddOverflow}^{2^i}_{q}\big) + \sum_{i=1}^{N} 2^{N-i} T\big(\mathrm{B2Abit}^{2^i}_{q'}\big)\,.$$

Using $T(\mathrm{SecAddOverflow}^{n'}_{q}) \lesssim 2T(\mathrm{SecAddBin}^{n'}_{w})$ [5, 16], we get

$$T\big(\mathrm{SecRescale}^{2^N}_{q,q',r}\big) \lesssim T\big(\mathrm{SecAddBin}^{2^{N-1}}_{w}\big) + T\big(\mathrm{SecAddBin}^{2^N}_{w}\big) + \sum_{i=1}^{N-1} 2^{N-i+1} T\big(\mathrm{SecAddBin}^{2^i}_{w}\big) + \sum_{i=1}^{N} 2^{N-i} T\big(\mathrm{B2Abit}^{2^i}_{q'}\big)\,,$$

We now determine the asymptotic complexity of our gadget. We use the complexities reported in the literature for the base gadgets: $T(\mathrm{SecAddBin}^{n'}_{2^w}) = O(wn'^2)$, as shown by Coron *et al.* [13], and $T(\mathrm{B2Abit}^{n'}_{q'}) = O(n'^2)$ using the gadget of Schneider *et al.* [24]. We thus obtain

$$\begin{aligned} T\big(\mathrm{SecRescale}^{2^N}_{q,q',r}\big) &= O\left(w(2^{N-1})^2 + w(2^N)^2 + \sum_{i=1}^{N-1} 2^{N-i+1} w(2^i)^2 + \sum_{i=1}^{N} 2^{N-i}(2^i)^2\right), \\ &= O\left(wn^2 + wn\sum_{i=1}^{N-1} 2^i + n\sum_{i=1}^{N} 2^i\right), \\ &= O(n^2 \log q). \end{aligned}$$

In comparison, the proposal of Coron *et al.* for high-order masked compression to d bits has a time complexity of $T(\mathrm{HOCompress}_{q,d}) = O(n^2(\log q + \log n))$ [12], which is (asymptotically) slightly worse than our result. This slight inefficiency of their solution comes from their use of a suboptimal representation for the intermediate values of compression: they use a fixed-point representation to calculate a fraction having a non–power-of-two denominator. Consequently, they can only represent approximations of this fraction, and require a growing precision for the approximation when the number of shares increases (hence the $\log n$ in the cost of their construction). In contrast, our representation of the intermediate values is exact, and remains so with any number of shares.

Table 1. Comparison of SecRescale with HOCompress. The "random" column reports the amount of randomness in bits. The "div. ops." column gives the number of division operations, and their size is expressed as "$N \div D \to Q(R)$", with N (resp. D, Q, R) the size in bits of the dividend (resp. divisor, quotient, remainder). The "arith. ops." column indicates the number of the other arithmetic operations, and "Bool. ops." the number of Boolean operations, where $N \times w$ means N Boolean operations on words of w bits (or on individual bits if $w = 1$).

n	d	algorithm	random	div. ops.	div. size	arith. ops.	Bool. ops.
2	4	SecRescale	164	2	$16 \div 12 \to 4(12)$	15	3390×1
		HOCompress	17	2	$29 \div 12 \to 17$	7	92×17
		HOCompress*	272	2	$29 \div 12 \to 17$	7	4452×1
2	11	SecRescale	178	2	$23 \div 12 \to 11(12)$	15	3390×1
		HOCompress	24	2	$36 \div 12 \to 24$	7	127×24
		HOCompress*	552	2	$36 \div 12 \to 24$	7	8974×1
3	4	SecRescale	824	3	$16 \div 12 \to 4(12)$	40	11992×1
		HOCompress	1044	3	$30 \div 12 \to 18$	10	590×18
		HOCompress*	1224	3	$30 \div 12 \to 18$	10	16483×1
3	11	SecRescale	880	3	$23 \div 12 \to 11(12)$	40	11992×1
		HOCompress	1975	3	$37 \div 12 \to 25$	10	814×25
		HOCompress*	2400	3	$37 \div 12 \to 25$	10	32170×1
4	4	SecRescale	1804	4	$16 \div 12 \to 4(12)$	69	23908×1
		HOCompress	2052	4	$30 \div 12 \to 18$	13	1066×18
		HOCompress*	2448	4	$30 \div 12 \to 18$	13	31024×1
4	11	SecRescale	1916	4	$23 \div 12 \to 11(12)$	69	23908×1
		HOCompress	3900	4	$37 \div 12 \to 25$	13	1472×25
		HOCompress*	4800	4	$37 \div 12 \to 25$	13	60592×1
7	4	SecRescale	7896	7	$16 \div 12 \to 4(12)$	231	90684×1
		HOCompress	11153	7	$31 \div 12 \to 19$	22	4553×19
		HOCompress*	11286	7	$31 \div 12 \to 19$	22	129028×1
7	11	SecRescale	8358	7	$23 \div 12 \to 11(12)$	231	90684×1
		HOCompress	20722	7	$38 \div 12 \to 26$	22	6212×26
		HOCompress*	21450	7	$38 \div 12 \to 26$	22	244640×1
8	4	SecRescale	10840	8	$16 \div 12 \to 4(12)$	298	121960×1
		HOCompress	14820	8	$31 \div 12 \to 19$	25	5912×19
		HOCompress*	15048	8	$31 \div 12 \to 19$	25	169608×1
8	11	SecRescale	11456	8	$23 \div 12 \to 11(12)$	298	121960×1
		HOCompress	27560	8	$38 \div 12 \to 26$	25	8068×26
		HOCompress*	28600	8	$38 \div 12 \to 26$	25	321648×1

A refined cost comparison would need to consider the running time on an actual device, but we can already give some rough results by simply counting the number of operations. In Table 1, we report the number of operations of SecRescale (including the optimization described in Section 4.2) and HOCompress for $d \in \{4, 11\}$ and various values of n. The table also reports on the number of random bits necessary. We have divided the operation count into three categories: the divisions, the (other) arithmetic operations and the Boolean operations[7].

- The number of divisions is n in all the cases. However, their precision differs, as noted in the table.
- The number of arithmetic operations accounts for additions and multiplications, without further refinements. As noted above, the precision of these operations vary, on $\log_2(qq')$ or $\log_2 q'$ bits for SecRescale, and on $d + \alpha = \log_2(qq'n)$ bits for HOCompress, but our count does not distinguish them.
- The Boolean operations include AND, XOR and NOT operations, either in a word-oriented bitwise fashion or in a bitsliced mode. For HOCompress, the arithmetic-to-Boolean conversion used in Algorithm 1 is due to Coron et al. [13, Algorithm 4], with Goubin's conversion [18] for the base case $n = 2$, and Goubin's SecAdd [13, Algorithm 3] for all n; this method uses Boolean operations on words of $d+\alpha$ bits. On the other hand, SecRescale uses bitsliced gadgets, and in that case we count the number of bit-level operations (which may be grouped per w bits on w-bit processors). As an indicative reference, we define HOCompress* as Algorithm 1 with the arithmetic-to-Boolean conversion of Bronchain and Cassiers [7, Algorithm 8], so that it uses the same bit-oriented SecAdd as our proposal – note, however, that this is somewhat less efficient than the original HOCompress.

We can see that with these metrics, HOCompress is more efficient for $n = 2$, thanks to Goubin's very fast first-order A2B conversion. However, for $n \geqslant 3$, our solution performs significantly better than HOCompress in terms of the randomness requirements. Still for $n \geqslant 3$, the most efficient method in terms of single-bit Boolean operations depends on d: for $d = 4$, SecRescale is less efficient than HOCompress by 5 % to 20 %, while for $d = 11$, SecRescale is more efficient by 35 % to 44 %. The only metric in which our solution is constantly more expensive is in the number of arithmetic operations (apart from divisions). We caution the reader, however, that these figures can only give an approximate idea of a cost comparison, since they ignore hidden costs such as, when applicable, the conversions between canonical and bitsliced representations of Boolean sharings [7]: these are impossible to estimate generically (such hidden factors should equally affect SecRescale and HOCompress*, but not necessarily HOCompress).

As an additional advantage, our solution is generic: while ML-KEM compression, and the masked gadget of Coron *et al.*, always compress to a power-of-two range $q' = 2^d$, this restriction does not exist for SecRescale. Any integer $q' \geqslant 2$

[7] Coron et al.'s performance figures for HOCompress [12, Table 8] can be approximately recovered from our table as random/$(d + \alpha)$ + div. ops. + arith. ops. + Bool. ops.

can be chosen, including $q' > q$, in which case the rescaling is more akin to ML-KEM decompression.

A notable difference between SecRescale and HOCompress is that, while our gadget outputs its result as an arithmetic sharing modulo q', that of Coron *et al.* instead outputs a Boolean sharing. The choice between the two gadgets may thus be additionally determined by the type of operations to be performed on the result of compression. We note that adding an A2B or B2A conversion to the output of SecRescale or HOCompress, so that they have the same kind of output sharing, would have no influence on the comparison of their *asymptotic* complexities; their practical cost, however, would certainly be affected by a significant amount, depending on the parameters of the gadgets.

5 Conclusions

In this paper, we have described a novel way of computing ML-KEM ciphertext compression securely, by viewing it as a more generic integer rescaling. We implement this rescaling with a share-wise approximation step and a nonlinear correction step, and this gives us a secure algorithm whose data width is independent of the masking order. This allows us to reuse the secure-addition and masking-conversion gadgets that are already employed in other parts of the secure implementation of ML-KEM. In addition, our proposal has slightly better asymptotic complexity than achieved in previous works; for the second and higher masking orders, it uses fewer fresh random bits than the state of the art, and also fewer single-bit logical operations for large enough d. This makes us believe that it will perform better in a significant proportion of practical implementations.

As future work, it would be beneficial to refine this comparison by analyzing the actual runtime of these solutions on a defined software or hardware platform, as well as evaluating the actual security of a full implementation.

A Deferred proofs

A.1 Proof of Lemma 2

Proof. J is initialized to n copies of 1. It is then only updated at Line 17, where its terms are summed in non-overlapping pairs (if there are an odd number of terms, the last one is left unchanged), preserving the overall sum. □

A.2 Proof of Lemma 3

Proof. We have

$$\begin{aligned}\bigoplus_{\boldsymbol{k}=\boldsymbol{j}}^{\boldsymbol{j'}} f^{\boldsymbol{k}} + q\sum_{\boldsymbol{k}=\boldsymbol{j}}^{\boldsymbol{j'}} c^{\boldsymbol{k}} &\equiv \bigoplus_{\boldsymbol{k}=\boldsymbol{j}}^{\boldsymbol{j'}} f^{\boldsymbol{k}} + q\bigoplus_{\boldsymbol{k}=\boldsymbol{j}}^{\boldsymbol{j'}} b^{\boldsymbol{k}} \equiv \left(\bigoplus_{\boldsymbol{k}=\boldsymbol{j}}^{\boldsymbol{j'}} F_0^{\boldsymbol{k}}\right) + \left(\bigoplus_{\boldsymbol{k}=\boldsymbol{j}}^{\boldsymbol{j'}} F_1^{\boldsymbol{k}}\right) \pmod{qq'} \\ &\equiv \mathrm{HybridUnmask}((J_{\mathrm{before},2i}, J_{\mathrm{before},2i+1}), f_{\mathrm{before}}^{\boldsymbol{j}:\boldsymbol{j'}}) \pmod{qq'}\end{aligned}$$

□

A.3 Proof of Lemma 4

Proof. First note that it is legal to compute HybridUnmask$(J, f^\star)$ since, by Lemma 2, J always sums to n, and $f^\star$ is an n-tuple.

Before the first iteration of the outer loop, the property is satisfied since by Eq. (4),

$$\text{HybridUnmask}(J, f^\star) + q\sum_{\boldsymbol{k}=0}^{n-1} y^{\boldsymbol{k}} = \sum_{\boldsymbol{k}=0}^{n-1}(f^{\boldsymbol{k}} + qy^{\boldsymbol{k}}) \equiv \sum_{\boldsymbol{k}=0}^{n-1}(x^{\boldsymbol{k}}q' + r_{\boldsymbol{k}}) \pmod{qq'}.$$

At an arbitrary iteration of the loop, denote with subscript $_{\text{before}}$ the value of variables immediately before this iteration, and without subscript, the values immediately after this iteration. Assume the property holds before the iteration. By summing both sides of the equivalence in Lemma 3 over all successive values of $(\boldsymbol{j}, \boldsymbol{j'})$ at Line 7, we get

$$\text{HybridUnmask}(J, f^\star) + q\sum_{\boldsymbol{k}=0}^{n-1} c^{\boldsymbol{k}} \equiv \text{HybridUnmask}(J_{\text{before}}, f^\star_{\text{before}}) \pmod{qq'}.$$

Adding $q\sum_{\boldsymbol{k}=0}^{n-1} y^{\boldsymbol{k}}_{\text{before}}$ to each side, and applying the induction hypothesis, we get

$$\text{HybridUnmask}(J, f^\star) + q\left(\sum_{\boldsymbol{k}=0}^{n-1} c^{\boldsymbol{k}} + \sum_{\boldsymbol{k}=0}^{n-1} y^{\boldsymbol{k}}_{\text{before}}\right) \equiv \sum_{\boldsymbol{i}=0}^{n-1}(x^{\boldsymbol{i}}q' + r_{\boldsymbol{i}}) \pmod{qq'},$$

from which we get the expected equivalence since $y^\star = c^\star + y^\star_{\text{before}}$. □

A.4 Proof of Theorem 1

Proof. Before the first iteration, $|J| = n$. At each iteration, Line 17 halves the length of J, rounding up. Thus, the loop stops after k iterations, k being the least integer such that $\lceil n/2^k \rceil = 1$.

Due to the condition of the outer loop, $|J| \leqslant 1$ once the algorithm has terminated. Since by Lemma 2, J is a composition of n, we conclude that it is the singleton (n) at this point. Applying Lemma 4 there, we deduce that

$$\text{HybridUnmask}((n), f^\star) + q\sum_{\boldsymbol{i}=0}^{n-1} y^{\boldsymbol{i}} \equiv \sum_{\boldsymbol{i}=0}^{n-1}(x^{\boldsymbol{i}}q' + r_{\boldsymbol{i}}) \pmod{qq'},$$

or equivalently,

$$\bigoplus_{\boldsymbol{i}=0}^{n-1} f^{\boldsymbol{i}} + q\sum_{\boldsymbol{i}=0}^{n-1} z^{\boldsymbol{i}} \equiv q'\left(\sum_{\boldsymbol{i}=0}^{n-1} x^{\boldsymbol{i}}\right) + r \pmod{qq'}.$$

As $f^\star$ is the sum output of the $\text{SecAddOverflow}_q^n$ gadget, $\bigoplus_{\boldsymbol{i}=0}^{n-1} f^{\boldsymbol{i}} < q$. Thus, the floor division by q of each side of the previous equation gives the sought result,

$$\sum_{\boldsymbol{i}=0}^{n-1} z^{\boldsymbol{i}} \equiv \left\lfloor \frac{q'(\sum_{\boldsymbol{i}=0}^{n-1} x^{\boldsymbol{i}}) + r}{q} \right\rfloor \pmod{q'}. \qquad \square$$

References

1. Assael, G.: Hardware Acceleration of Post-Quantum Cryptography for Embedded Systems. Ph.D. thesis, Université Grenoble Alpes (Dec 2024), https://theses.hal.science/tel-05127604
2. Assael, G., Elbaz-Vincent, P.: Provably secure and area-efficient modular addition over boolean shares. IACR Communications in Cryptology **1**(2) (2024). https://doi.org/10.62056/aee0zoja5
3. Assael, G., Van Assche, G.: Compression masquée. French patent FR3160834A1 (2024)
4. Bache, F., Güneysu, T.: Boolean masking for arithmetic additions at arbitrary order in hardware. Applied Sciences **12**(5) (2022). https://doi.org/10.3390/app12052274
5. Barthe, G., Belaïd, S., Espitau, T., Fouque, P.A., Grégoire, B., Rossi, M., Tibouchi, M.: Masking the GLP lattice-based signature scheme at any order. In: Nielsen, J.B., Rijmen, V. (eds.) EUROCRYPT 2018, Part II. LNCS, vol. 10821, pp. 354–384. Springer, Cham (Apr / May 2018). https://doi.org/10.1007/978-3-319-78375-8_12
6. Bos, J.W., Gourjon, M., Renes, J., Schneider, T., van Vredendaal, C.: Masking Kyber: First- and higher-order implementations. IACR TCHES **2021**(4), 173–214 (2021). https://doi.org/10.46586/tches.v2021.i4.173-214, https://tches.iacr.org/index.php/TCHES/article/view/9064
7. Bronchain, O., Cassiers, G.: Bitslicing arithmetic/boolean masking conversions for fun and profit with application to lattice-based KEMs. IACR TCHES **2022**(4), 553–588 (2022). https://doi.org/10.46586/tches.v2022.i4.553-588
8. Cassiers, G., Standaert, F.X.: Provably secure hardware masking in the transition- and glitch-robust probing model: Better safe than sorry. IACR TCHES **2021**(2), 136–158 (2021). https://doi.org/10.46586/tches.v2021.i2.136-158, https://tches.iacr.org/index.php/TCHES/article/view/8790
9. Cassiers, G., Grégoire, B., Levi, I., Standaert, F.X.: Hardware private circuits: From trivial composition to full verification. IEEE Transactions on Computers **70**(10), 1677–1690 (2021). https://doi.org/10.1109/TC.2020.3022979
10. Cassiers, G., Standaert, F.X.: Trivially and efficiently composing masked gadgets with Probe Isolating Non-Interference. IEEE Transactions on Information Forensics and Security **15**, 2542–2555 (2020). https://doi.org/10.1109/TIFS.2020.2971153
11. Chari, S., Jutla, C.S., Rao, J.R., Rohatgi, P.: Towards sound approaches to counteract power-analysis attacks. In: Wiener, M.J. (ed.) CRYPTO'99. LNCS, vol. 1666, pp. 398–412. Springer, Berlin, Heidelberg (Aug 1999). https://doi.org/10.1007/3-540-48405-1_26
12. Coron, J.S., Gérard, F., Montoya, S., Zeitoun, R.: High-order polynomial comparison and masking lattice-based encryption. IACR TCHES **2023**(1), 153–192 (2023). https://doi.org/10.46586/tches.v2023.i1.153-192

13. Coron, J.S., Großschädl, J., Vadnala, P.K.: Secure conversion between Boolean and arithmetic masking of any order. In: Batina, L., Robshaw, M. (eds.) CHES 2014. LNCS, vol. 8731, pp. 188–205. Springer, Berlin, Heidelberg (Sep 2014). https://doi.org/10.1007/978-3-662-44709-3_11
14. Faust, S., Grosso, V., Pozo, S.M.D., Paglialonga, C., Standaert, F.X.: Composable masking schemes in the presence of physical defaults & the robust probing model. IACR TCHES **2018**(3), 89–120 (2018). https://doi.org/10.13154/tches.v2018.i3.89-120, https://tches.iacr.org/index.php/TCHES/article/view/7270
15. Module-lattice-based key-encapsulation mechanism standard. National Institute of Standards and Technology, NIST FIPS PUB 203, U.S. Department of Commerce (Aug 2024). https://doi.org/10.6028/NIST.FIPS.203
16. Fritzmann, T., Beirendonck, M.V., Roy, D.B., Karl, P., Schamberger, T., Verbauwhede, I., Sigl, G.: Masked accelerators and instruction set extensions for post-quantum cryptography. IACR TCHES **2022**(1), 414–460 (2022). https://doi.org/10.46586/tches.v2022.i1.414-460
17. Fujisaki, E., Okamoto, T.: Secure integration of asymmetric and symmetric encryption schemes. In: Wiener, M.J. (ed.) CRYPTO'99. LNCS, vol. 1666, pp. 537–554. Springer, Berlin, Heidelberg (Aug 1999). https://doi.org/10.1007/3-540-48405-1_34
18. Goubin, L.: A sound method for switching between Boolean and arithmetic masking. In: Koç, Çetin Kaya., Naccache, D., Paar, C. (eds.) CHES 2001. LNCS, vol. 2162, pp. 3–15. Springer, Berlin, Heidelberg (May 2001). https://doi.org/10.1007/3-540-44709-1_2
19. Ishai, Y., Sahai, A., Wagner, D.: Private circuits: Securing hardware against probing attacks. In: Boneh, D. (ed.) CRYPTO 2003. LNCS, vol. 2729, pp. 463–481. Springer, Berlin, Heidelberg (Aug 2003). https://doi.org/10.1007/978-3-540-45146-4_27
20. Kocher, P.C., Jaffe, J., Jun, B.: Differential power analysis. In: Wiener, M.J. (ed.) CRYPTO'99. LNCS, vol. 1666, pp. 388–397. Springer, Berlin, Heidelberg (Aug 1999). https://doi.org/10.1007/3-540-48405-1_25
21. Oder, T., Schneider, T., Pöppelmann, T., Güneysu, T.: Practical CCA2-secure masked Ring-LWE implementations. IACR TCHES **2018**(1), 142–174 (2018). https://doi.org/10.13154/tches.v2018.i1.142-174, https://tches.iacr.org/index.php/TCHES/article/view/836
22. Reparaz, O., Roy, S.S., Vercauteren, F., Verbauwhede, I.: A masked ring-LWE implementation. In: Güneysu, T., Handschuh, H. (eds.) CHES 2015. LNCS, vol. 9293, pp. 683–702. Springer, Berlin, Heidelberg (Sep 2015). https://doi.org/10.1007/978-3-662-48324-4_34
23. Schneider, T., Moradi, A., Güneysu, T.: Arithmetic addition over Boolean masking - towards first- and second-order resistance in hardware. In: Malkin, T., Kolesnikov, V., Lewko, A.B., Polychronakis, M. (eds.) ACNS 15International Conference on Applied Cryptography and Network Security. LNCS, vol. 9092, pp. 559–578. Springer, Cham (Jun 2015). https://doi.org/10.1007/978-3-319-28166-7_27
24. Schneider, T., Paglialonga, C., Oder, T., Güneysu, T.: Efficiently masking binomial sampling at arbitrary orders for lattice-based crypto. In: Lin, D., Sako, K. (eds.) PKC 2019, Part II. LNCS, vol. 11443, pp. 534–564. Springer, Cham (Apr 2019). https://doi.org/10.1007/978-3-030-17259-6_18
25. Shor, P.W.: Polynominal time algorithms for discrete logarithms and factoring on a quantum computer. In: Adleman, L.M., Huang, M.A. (eds.) Algorithmic Number Theory, First International Symposium, ANTS-I, Ithaca, NY, USA, May 6-9, 1994, Proceedings. Lecture Notes in Computer Science, vol. 877, p. 289. Springer (1994). https://doi.org/10.1007/3-540-58691-1_68, https://doi.org/10.1007/3-540-58691-1_68