IACR Transactions on Cryptographic Hardware and Embedded Systems
ISSN 2569-2925, Vol. 2023, No. 2, pp. 180–211. DOI:10.46586/tches.v2023.i2.180-211

High-order masking of NTRU

Jean-Sébastien Coron[1], François Gérard[1], Matthias Trannoy[1,2] and Rina Zeitoun[2]

[1] University of Luxembourg, Luxembourg
jean-sebastien.coron@uni.lu,francois.gerard@uni.lu
[2] IDEMIA, Cryptography & Security Labs, Courbevoie, France
matthias.trannoy@idemia.com,rina.zeitoun@idemia.com

Abstract. The main protection against side-channel attacks consists in computing every function with multiple shares via the masking countermeasure. While the masking countermeasure was originally developed for securing block-ciphers such as AES, the protection of lattice-based cryptosystems is often more challenging, because of the diversity of the underlying algorithms. In this paper, we introduce new gadgets for the high-order masking of the NTRU cryptosystem, with security proofs in the classical ISW probing model. We then describe the first fully masked implementation of the NTRU Key Encapsulation Mechanism submitted to NIST, including the key generation. To assess the practicality of our countermeasures, we provide a concrete implementation on ARM Cortex-M3 architecture, and eventually a t-test leakage evaluation.

Keywords: High-order masking · lattice-based cryptography · NTRU

1 Introduction

Post-quantum cryptography. The `RSA` and `ECC` cryptosystems rely on the hardness of the integer factorization and the discrete logarithm problems respectively. These problems, which we can assume to be hard on a classical computer, are however vulnerable to a quantum one. Peter Shor in 1995 has indeed designed an algorithm running on a quantum computer that ensures a polynomial-time solution. In light of these new threats, the National Institute of Standards and Technology (NIST) initiated in 2016 a standardization process for post-quantum cryptography that has reached its last round.

The `NTRU` cryptosystem. The `NTRU` cryptosystem was introduced in 1996 by Hoffstein, Pipher and Silverman [HPS98] covering both encryption and signature. Its security relies on the problem of finding small solutions to a system of linear equations over polynomial rings, which is assumed to remain hard even in the presence of a quantum computer. Therefore, it is closely related to the Shortest and Closest Vector Problems (SVP/CVP) in lattices. Despite not being equivalent neither to SVP nor CVP, the `NTRU` cryptosystem nonetheless resisted more than two decades of cryptanalysis. Moreover variants of `NTRU` were proven to be secure in the (Quantum) Random Oracle Model under the Ring Learning With Error (R-LWE) hardness assumption [SS13]. In terms of performance, `NTRU` is known to be currently one of the fastest public key cryptosystem altogether with moderate key-size, making it a reasonable choice for embedded cryptography. Its performance granted it several standards, *e.g.*, IEEE Std 1363.1, X9.98 and PQCRYPTO. Recently `NTRU` was one of the finalists of the NIST post-quantum cryptography standardization effort; the `Kyber` algorithm has finally been selected for standardization.

Received: 2022-10-15 Accepted: 2022-12-15 Published: 2023-03-06

Side-channel attacks and the masking countermeasure. As for any other cryptosystem, an NTRU implementation on embedded device is vulnerable to side-channel attacks. These attacks exploit physical leakages happening during the execution of the algorithm to recover the key. We refer to [PPM17, HCY20, XPRO20, EMVW22] for examples of such attacks. Side-channel attacks can be prevented by using the masking countermeasure. It consists in splitting each secret variable into shares, for $x = x_1 \oplus \cdots \oplus x_n$ with Boolean masking. Then by processing each share independently, any leakage on at most $n-1$ shares x_i will not reveal information about the secret x. Formally, in this paper we consider the classical probing model introduced in [ISW03], with an attacker being able to probe any set of at most t variables in the circuit. The authors showed that using at least $n = 2t+1$ shares, one can transform any Boolean circuit C into a circuit C' of size $\mathcal{O}(|C| \cdot t^2)$, such that an adversary with t probes on C' is no more powerful that an adversary with no probe at all. Later, finer notions of security were formalized by Barthe *et al.* in [BBD+16], who introduced the notions of (Strong) Non-Interference NI/SNI. This enables to reach t-probing security with $n = t + 1$ shares only, via a composition theorem.

While any encryption scheme can be written as a Boolean circuit and then protected using the above transform, in practice that would be quite inefficient. Indeed, lattice-based cryptography usually requires to perform both Boolean and arithmetic operations, and moreover, the NTRU cryptosystem combines arithmetic operations modulo $q = 2^k$ and modulo 3. It is therefore more efficient to mask some intermediate variables with arithmetic masking modulo q or modulo 3, instead of with Boolean masks only. One must therefore repeatedly convert between these masked representations.

The first conversions between Boolean and arithmetic masking were described in [Gou01] for first-order security. It was then generalized to higher order in [CGV14], with complexity $\mathcal{O}(n^2 \cdot k)$ for n shares and k-bit words. Recently, a generic conversion algorithm was described in [CGMZ22], based on table-recomputation. It allows to high-order compute any function $f : G \to H$ between two groups G and H, with complexity $\mathcal{O}(|G| \cdot n^2)$. For example, by taking $G = \mathbb{Z}_3$ and $H = \mathbb{Z}_q$, one can efficiently convert from arithmetic masking modulo 3 to arithmetic masking modulo q, which will be useful in the context of NTRU.

Masking lattice-based public-key encryption. We review the existing masked implementations of lattice-based public-key encryption, including the NIST finalists Kyber, Saber and NTRU. To achieve IND-CCA security, the Kyber and Saber schemes use the Fujisaki-Okamoto transformation [FO99], based on the recomputation and comparison of the ciphertext during decryption. The first completely masked implementation of Kyber secure against high-order attacks was described in [BGR+21]. For the ciphertext comparison, the masked recomputed ciphertext remains in uncompressed form, so that the compression function from Kyber need not be high-order masked. Alternative techniques for performing the ciphertext comparison have also been recently described in [CGMZ21], for both Kyber and Saber.

However, the CCA security of NTRU in the NIST submission [CDH+19] does not rely on the FO transform, but rather on the membership of the message to a specific space set. This is to ensure the well-formedness of the ciphertext, based on the correctness of the underlying deterministic PKE scheme [BP18]. Formally, the CCA security follows from the property that for $(r, m) \in \mathcal{L}_r \times \mathcal{L}_m$, where $\mathcal{L}_r$ and $\mathcal{L}_m$ represent the plaintext space sets:

$$\texttt{NTRU.Enc}((r, m), pk) = c \Leftrightarrow \texttt{NTRU.Dec}(c, sk) = (r, m)$$

Therefore, the well-formedness of c is ensured by membership the test $(r, m) \overset{?}{\in} \mathcal{L}_r \times \mathcal{L}_m$.

So far in the literature the only masked implementation of NTRU is provided by [SMS19], for security against CPA and first-order attacks only. The authors focus on protecting the polynomial product $c \cdot f \bmod q$, for the ciphertext c and the private-key f. Recently, [REB+21] introduced a generic side-channel CCA against NTRU exploiting the leakage during the membership test $(r, m) \in \mathcal{L}_r \times \mathcal{L}_m$. This demonstrates that a masked implementation must include the masking of this membership test.

More recently, in a concurrent work [KLRBG22], the authors described a high-order masked algorithm to perform the polynomial inversion in the key generation of NTRU, based on a conversion from additive to multiplicative masking. The authors claimed that their high-order conversion algorithm can achieve arbitrary-order security, but without a security proof. As a security evaluation, the authors used a common fixed vs. random univariate first-order Test Vector Leakage Assessment (TVLA) evaluation procedure, with 100 000 power traces. However, we show in this paper that their algorithm is actually insecure: we exhibit a 3-rd order attack for any number of shares n in the countermeasure (see Section 5.1). We then describe a repaired algorithm with a proof of security in the ISW probing model.

Our contributions. In this paper we provide the first high-order masking of the NTRU KEM finalist. More precisely, we provide a full high-order masking of both the Decapsulate algorithm (for IND-CCA decryption), and the key generation algorithm. We consider the two HPS and HRSS variants of the NTRU submission [CDH+19]. Our countermeasures are proven secure in the classical ISW probing model, using the NI/SNI methodology.

We argue that key generation must also be protected against side-channel attacks, because in practice, the key generation procedure can be performed directly in the embedded platform, and template attacks can be quite effective against key generation. To prove the side-channel resistance of KeyGen, we use the same ISW probing model as for other operations. That is, when using $n = t + 1$ shares, the KeyGen algorithm should be resistant against an adversary performing a t-th order probing attack.

Our techniques are as follows. For decryption, the main challenge is to compute the reduction modulo 3 of a polynomial a which is initially masked modulo $q = 2^k$. For this we proceed coefficient-wise by first converting the arithmetic sharing modulo q into Boolean shares, and then converting back to arithmetic modulo 3. We also describe the high-order masking of the membership tests $r \in \mathcal{L}_r$ and $m \in \mathcal{L}_m$. For the later, in the HPS version, one needs to check that the polynomial m has exactly $d/2$ coefficients equal to 1, and $d/2$ coefficients equal to -1, for $d = q/8 - 2$. For this, we high-order compute the sum of the coefficients and check that it is equal to 0 modulo q, and we check that the sum of the squares of the coefficients is equal to d modulo q.

For masking the key generation, we show how to mask the sampling of the private key, which includes the sampling of an arithmetically masked polynomial with exactly $d/2$ coefficients equal to 1 and exactly $d/2$ equal to -1. To do so, we start with a fixed polynomial g_I with the first $d/2$ coefficients equal to 1, the next $d/2$ coefficients equal to -1, and the remaining coefficients equal to 0. We then compute an arithmetic sharing $g_1, \ldots, g_n$ of g_I. We then repeat n times the following procedure: we generate a random permutation π of the coefficients and apply π to each share g_i, and then linearly refresh the shares g_i. Eventually, we return the shared polynomial $g_1, \ldots, g_n$. We show that we indeed obtain an n-sharing of a random polynomial g with the right distribution, and moreover an adversary with at most $n - 1$ probes learns nothing about the secret polynomial g.

For the key generation, we also show how to high-order compute the inverse of polynomials in $\mathbb{Z}_q[X]/(\Phi_\ell)$ and $\mathbb{Z}_3[X]/(\Phi_\ell)$. In the NIST submission, these inverses are computed using the almost inverse algorithm. However, such method would be quite challenging to mask, therefore we use exponentiation algorithms instead. More precisely, we compute the inverse of an element x in $\mathbb{Z}_2[X]/\Phi_\ell$ by using the relation $x^{-1} = x^{2^{\ell-1}-2}$. Thanks to the linearity of the square in characteristic 2, such exponentiation only requires $\mathcal{O}(\log \ell)$ multiplications, instead of $\mathcal{O}(\ell)$. One can then lift the inverse from modulo 2 to modulo 2^k. Both operations are easy to high-order mask with n shares, and as previously, we prove that an adversary with at most $n-1$ probes learns nothing about the secret-key. We then provide a comparison with our repaired algorithm from [KLRBG22].

Finally, using the above gadgets, we describe a full high-order masking of both the Decapsulate algorithm (for IND-CCA decryption), and of the key generation algorithm. For Decapsulate, this includes the masking of the PackS3 algorithm for converting ternary polynomials into a sequence of bytes. Namely, the PackS3 algorithm is used for computing the hash $k_1 = H_1(r, m)$ when recovering the session key k_1, which must be output in masked Boolean form.

Implementation. In order to assess the practicality of our countermeasures, we have performed a proof of concept implementation in C of the fully masked Decapsulate and KeyGen. We have run our implementation on a laptop equipped with an Intel CPU, and also on a Cortex-M3 core mounted on an Arduino Due board. We provide the performance analysis in Section 8. The source code can be found at

https://github.com/fragerar/Masked_NTRU

Finally, we have performed a leakage evaluation with a fixed vs random t-test over 10 000 traces for one of the main gadgets, namely the reduction modulo 3 used in Decapsulate. For this, we have used the ChipWhisperer Lite board embedding a Cortex-M4 microcontroller (STM32F303) and a light oscilloscope; we provide the results in Section 8.

2 Notations and security definitions

2.1 Notations

Integer ring. Let q be an integer, $\mathbb{Z}_q$ will denote the ring of integers modulo q. Depending on the context we will need to switch between two equivalent representations of the ring $\mathbb{Z}_q$: positive representation $\mathbb{Z}_q \simeq \{0, 1, \ldots, q-1\}$, and centered representation, $\mathbb{Z}_q \simeq \{-q/2+1, \ldots, 0, \ldots, q/2\}$ for even q, and $\mathbb{Z}_q \simeq \{-(q-1)/2, \ldots, 0, \ldots, (q-1)/2\}$ for odd q.

For any integer x, $x \bmod q$ will denote the positive representative of x, and $x \bmod^{\pm} q$ the centered one. We denote by $x \gg k$ (resp. $x \ll k$) the right (resp. left) shifting of an integer x by k positions, equivalently $x \gg k = \lfloor x/2^k \rfloor$ (resp. $x \ll k = x \cdot 2^k$).

Polynomial ring. Let q be an integer, we denote by $\mathbb{Z}_q[X]$ the ring of polynomials with coefficient in $\mathbb{Z}_q$. For a prime ℓ, we let Φ_1 and Φ_ℓ be the first and the ℓ-th cyclotomic polynomials $X-1$ and $1+X+\cdots+X^{\ell-1}$ respectively.

We recall the notations from [CDH+19]. We denote by S/q the quotient ring $\mathbb{Z}_q[X]/\Phi_\ell$. A polynomial in $\mathbb{Z}[X]$ is said to be *ternary* if its coefficients are in $\{-1, 0, 1\}$. We denote by $\mathcal{T}$ the set of non-zero ternary polynomials of degree at most $\ell-2$. Equivalently, $\mathcal{T}$ can be seen as the set of representatives of non-zero polynomials from the quotient $\mathbb{Z}_3[X]/\Phi_\ell$. For an even positive integer d, we also denote by $\mathcal{T}(d)$ the subset of $\mathcal{T}$ consisting of polynomials that have exactly $d/2$ coefficients equal to $+1$ and $d/2$ coefficients equal to -1. Finally, let $\mathcal{T}_+$ denote the set of positively correlated ternary polynomials, *i.e* polynomials $v \in \mathcal{T}$ such that $\sum_i v_i \cdot v_{i+1} \geq 0$.

2.2 Definitions

We recall the definitions of (strong) non-interference security (SNI/NI) introduced in [BBD+16]. Thanks to these definitions, a proof of security against an attacker with at most t probes can proceed in two steps: firstly one proves that every gadget satisfies the SNI definition, secondly one applies a composition theorem. The SNI definition is stronger than NI in that the number of input shares needed for the simulation only depends on the number of internal probes and not on the number of output variables to be simulated. Fortunately, the NI definition is not restrictive since composing a NI gadget with an SNI one achieves SNI security. Hence, any NI gadget can be enhanced to SNI by applying an SNI mask refreshing to its output. In this paper, we will prove that all our gadgets achieve at least NI security.

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

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

3 The NTRU cryptosystem

In this section, we recall the second round `NTRU` submission from [CDH+19]. It is based on a deterministic public-key encryption scheme (DPKE) described in algorithms 1, 2 and 3. The Key Encapsulation Mechanism (KEM) is depicted in algorithms 4, 5 and 6. For the two submitted versions of `NTRU`, namely `NTRU-HPS` and `NTRU-HRSS`, we recall in Table 1 the definition of the sets of integer polynomials $\mathcal{L}_f$, $\mathcal{L}_g$, $\mathcal{L}_r$, $\mathcal{L}_m$, and the embedding Lift. For simplicity, the algorithms are described according to the `NTRU-HPS` version, for which $\mathsf{Lift}(m) = m$. We also recall in Table 2 the values of the parameter ℓ and modulus q for the four versions of `NTRU`.

Alg. 1 $\mathsf{KeyGen}(seed)$

$f \leftarrow \mathcal{L}_f$, $g \leftarrow \mathcal{L}_g$
$f_q \leftarrow (1/f) \bmod (q, \Phi_\ell)$
$h \leftarrow (3 \cdot g \cdot f_q) \bmod (q, \Phi_1\Phi_\ell)$
$h_q \leftarrow (1/h) \bmod (q, \Phi_\ell)$
$f_p \leftarrow (1/f) \bmod (3, \Phi_\ell)$
return $((f, f_p, h_q), h)$

Alg. 2 $\mathsf{Encrypt}(h, (r, m))$

$c \leftarrow r \cdot h + m \bmod (q, \Phi_1\Phi_\ell)$
return c

Alg. 3 $\mathsf{Decrypt}((f, f_p, h_q), c)$

if $c \neq 0 \bmod (q, \Phi_1)$ **return** $(0, 0, 1)$
$a \leftarrow (c \cdot f) \bmod (q, \Phi_1\Phi_\ell)$
$m \leftarrow (a \cdot f_p) \bmod (3, \Phi_\ell)$
$r \leftarrow ((c - m) \cdot h_q) \bmod (q, \Phi_\ell)$
if $(r, m) \in (\mathcal{L}_r, \mathcal{L}_m)$ **return** $(r, m, 0)$
else return $(0, 0, 1)$

Alg. 4 $\mathsf{KeyGen}'(seed)$

$((f, f_p, h_q), h) \leftarrow \mathsf{KeyGen}(seed)$
$s \leftarrow \{0,1\}^{256}$
return $((f, f_p, h_q, s), h)$

Alg. 5 $\mathsf{Encapsulate}(h)$

$coins \leftarrow \{0,1\}^{256}$
$(r, m) \leftarrow \mathsf{Samplerm}(coins)$
$c \leftarrow \mathsf{Encrypt}(h, (r, m))$
$k \leftarrow H_1(r, m)$
return (c, k)

Alg. 6 $\mathsf{Decapsulate}((f, f_p, h_q, s), c)$

$(r, m, fail) \leftarrow \mathsf{Decrypt}((f, f_p, h_q), c)$
$k_1 \leftarrow H_1(r, m)$
$k_2 \leftarrow H_2(s, c)$
if $fail = 0$ **return** k_1
else return k_2

Table 1: Definitions of polynomial sets and lifting application for `NTRU-HPS` and `NTRU-HRSS`.

	$\mathcal{L}_f$	$\mathcal{L}_g$	$\mathcal{L}_r$	$\mathcal{L}_m$	Lift
`HPS`	$\mathcal{T}$	$\mathcal{T}(q/8-2)$	$\mathcal{T}$	$\mathcal{T}(q/8-2)$	$m \mapsto m$
`HRSS`	$\mathcal{T}_+$	$\Phi_1 \cdot \mathcal{T}_+$	$\mathcal{T}$	$\mathcal{T}$	$m \mapsto \Phi_1 \cdot (m/\Phi_1 \bmod^{\pm} (3, \Phi_\ell))$

Table 2: Values of ℓ and q for the four versions of NTRU.

	ntruhps2048509	ntruhps2048677	ntruhps4096821	ntruhrss701
ℓ	509	677	821	701
q	2048	2048	4096	8192

The `NTRU` DPKE scheme. We briefly explain why the DPKE scheme works (alg. 1, 2, 3). Since ℓ is a prime, and 2 is of order $\ell-1$ in $\mathbb{Z}_\ell^*$, we get that Φ_ℓ is an irreducible polynomial modulo 2. We deduce that the set of polynomials modulo 2 and Φ_ℓ is a field, and therefore $f \in \mathcal{L}_f$ is invertible modulo 2 and Φ_ℓ. One can then lift the inverse from modulo 2 to modulo q and Φ_ℓ. The same holds for the inverse of f modulo 3. Note that from $g \in \mathcal{L}_g$, we have $g = 0 \pmod{q, \Phi_1}$, and therefore $h = 0 \pmod{q, \Phi_1}$.

The encryption of m is given by:

$$c = r \cdot h + m \pmod{(q, \Phi_1\Phi_\ell)}$$

From Line 2 of Algorithm 3, we have:

$$a = c \cdot f = (r \cdot h + m) \cdot f \pmod{(q, \Phi_1\Phi_\ell)}$$

By definition we have $h \cdot f = 3 \cdot g \pmod{q, \Phi_\ell}$. This gives $a = 3 \cdot g \cdot r + m \cdot f \pmod{q, \Phi_\ell}$. Moreover, from $m = 0 \pmod{\Phi_1}$, we have $c = 0 \pmod{q, \Phi_1}$, and therefore $a = 0 \pmod{q, \Phi_1}$. Besides we have $g = m = 0 \pmod{q, \Phi_1}$, therefore we deduce:

$$a = 3 \cdot g \cdot r + m \cdot f \pmod{q, \Phi_1\Phi_\ell} \tag{1}$$

One can show that the equation also holds over $\mathbb{Z}$, not only modulo q. Namely, the polynomials g, r, m and f have small coefficients, therefore the equality holds over $\mathbb{Z}$ when we represent the polynomials modulo q with coefficients between $-q/2$ and $q/2$. This gives:

$$a = 3 \cdot g \cdot r + m \cdot f \pmod{\Phi_1\Phi_\ell} \tag{2}$$

We deduce that $a = m \cdot f \pmod{3, \Phi_1\Phi_\ell}$, and therefore $m \equiv a \cdot f_p \pmod{3, \Phi_\ell}$. Since $\deg m \leq \ell - 2$ and m is ternary, we must have $m = a \cdot f_p \pmod{3, \Phi_\ell}$, as computed in Line 3 of Algorithm 3. Finally, we have:

$$(c - m) \cdot h_q \equiv (r \cdot h) \cdot h_q \equiv r \pmod{(q, \Phi_\ell)}$$

and since $\deg(r) \leq \ell - 2$, we can recover r at Line 4 with $r = (c - m) \cdot h_q \bmod (q, \Phi_\ell)$.

CCA security of `NTRU`. The CCA security of `NTRU` is a consequence of its rigidity. The rigidity expresses as follow, for $(r, m) \in \mathcal{L}_r \times \mathcal{L}_m$:

$$\mathsf{Encrypt}(h, (r, m)) = c \Leftrightarrow \mathsf{Decrypt}((f, f_p, h_q), c) = (r, m)$$

Therefore, the FO transformation can be avoided by using the membership check $(r, m) \in \mathcal{L}_r \times \mathcal{L}_m$ since it ensures a correct ciphertext recomputation. Eventually, the rigidity is ensured by the choice of parameters in Table 1, see [BP18, HRSS17].

The NTRU KEM. The KEM version of NTRU proceeds similarly to the NTRU DPKE scheme (alg. 4, 5, 6). It adds a seed s to the secret key. This seed is used for implicit rejection during the decapsulation in order to preserve CCA security [BP18]. The Encapsulate algorithm samples r and m according to their space set and encrypts them into c. Then it hashes (r, m) into the session key k. Eventually, the Decapsulate algorithm decrypts the ciphertext c into $(r', m', fail)$. When no decryption failure occurs, the rigidity of the NTRU DPKE schemes ensures that r' and m' match the original r and m from encryption, which enables to recover the session key k.

4 New gadgets for high-order masking NTRU

In this section, we describe the high-order masking of the main components of the NTRU cryptosystem. We recall in Appendix A the main masking tools, such as arithmetic vs Boolean conversions, and zero-testing with Boolean or arithmetic shares.

4.1 Decryption: masking the reduction modulo 3

The polynomial a at Step 2 of Decrypt (Algorithm 3) is arithmetically masked modulo q, because the secret-key f is arithmetically masked modulo q. Namely, given as input the ciphertext c and the masked secret-key $f = f_1 + \cdots + f_n \pmod q$, we obtain:

$$a = c \cdot f = (c \cdot f_1) + \cdots + (c \cdot f_n) \quad (\text{mod } q, \Phi_1\Phi_\ell),$$

and letting $a_i = c \cdot f_i \bmod (q, \Phi_1\Phi_\ell)$, we obtain $a = a_1 + \cdots + a_n \pmod q$ as required.

The main difficulty is then to compute the polynomial a modulo $(3, \Phi_\ell)$, which corresponds to Step 3 of Decrypt. Namely the polynomial a satisfies

$$a = 3 \cdot g \cdot r + m \cdot f \quad (\text{mod } q, \Phi_1\Phi_\ell)$$

where the polynomials g, r, m and f have small coefficients, and therefore the equality

$$a = 3 \cdot g \cdot r + m \cdot f \quad (\text{mod } \Phi_1\Phi_\ell)$$

holds over the integers (not only modulo q). This enables to get rid of the $3 \cdot g \cdot r$ part by reduction modulo 3. One must therefore perform this operation while the polynomial a is arithmetically masked modulo q. Note that we cannot directly reduce each share a_i modulo 3 when a is arithmetically masked modulo q, as the reduction modulo 3 is not linear over the ring $\mathbb{Z}_q$.[1] This implies that a more complex technique is required.

For this, the idea is to first convert each coefficient of a from arithmetic masking modulo q into Boolean masking, and then perform a conversion from Boolean masking to arithmetic masking modulo 3. More precisely, we write $q = 2^k$ and we consider a coefficient $-2^{k-1} \leq x < 2^{k-1}$. We write $x = 3 \cdot u + v$ with $0 \leq v < 3$. Given as input an arithmetic sharing of x modulo 2^k, we must output an arithmetic sharing of v modulo 3. We write $x^{(j)}$ the j-th bit of $x \bmod 2^k$, so we can write:

$$x = -2^{k-1}x^{(k-1)} + \sum_{j=0}^{k-2} 2^j \cdot x^{(j)} = 3 \cdot u + v$$

and therefore we obtain the value of $v = x \bmod 3$ as a function of the bits $x^{(j)}$ of x:

$$v = (-2^{k-1} \bmod 3) \cdot x^{(k-1)} + \sum_{j=0}^{k-2} (2^j \bmod 3) \cdot x^{(j)} \pmod 3 \tag{3}$$

[1] Consider for example a masking with two shares x_1 and x_2 with $q = 256$, and let $x = x_1 + x_2 \bmod 256$, with $x_1 = 222$ and $x_2 = 57$, which gives $x = 23$. If we reduce x_1 and x_2 directly modulo 3, we obtain $(222 \bmod 3) + (57 \bmod 3) = 0 \pmod 3$, but on the other hand we have $x \bmod 3 = 2$. So reducing the shares modulo 3 directly would give an incorrect result.

We now explain how to high-order compute v modulo 3 from an arithmetic masking of x modulo $q = 2^k$. Taking $x = x_1 + \cdots + x_n \pmod q$ as input, we first perform an arithmetic to Boolean masking conversion, so we obtain $x = y_1 \oplus \cdots \oplus y_n$ with $y_i \in \{0,1\}^k$ for all $1 \le i \le n$. Letting $y_i^{(j)}$ be the j-th bit of y_i, we have $x^{(j)} = y_1^{(j)} \oplus \cdots \oplus y_n^{(j)}$ for all $0 \le j < k$. Therefore we perform a Boolean to arithmetic modulo 3 conversion of each $x^{(j)}$, which gives for all $0 \le j < k$:

$$x^{(j)} = y_1^{(j)} \oplus \cdots \oplus y_n^{(j)} = z_1^{(j)} + \cdots + z_n^{(j)} \pmod 3 \tag{4}$$

Eventually, we obtain by combining (3) and (4):

$$\begin{aligned} v &= (-2^{k-1} \bmod 3) \cdot \sum_{i=1}^{n} z_i^{(k-1)} + \sum_{j=0}^{k-2} (2^j \bmod 3) \sum_{i=1}^{n} z_i^{(j)} \pmod 3 \\ &= \sum_{i=1}^{n} \left(\sum_{j=0}^{k-2} (2^j \bmod 3) z_i^{(j)} - (2^{k-1} \bmod 3) z_i^{(k-1)} \right) \pmod 3 \end{aligned}$$

which gives an n-sharing of v modulo 3, as required. We provide the corresponding algorithm below. We refer to Appendix A.1 for an overview of the conversion algorithms AtoB_{2^k} and BtoA_3, which are assumed to satisfy the SNI property. Note that our algorithm can work for any modulus q, not only 2^k, by using an algorithm for converting from arithmetic modulo q to Boolean masking at Line 1.

Algorithm 7 $\mathsf{Mod3Red}(v_1, \ldots, v_n)$

Input: An arithmetic sharing modulo 2^k $(x_1, \ldots, x_n)$ of $x \in [-2^{k-1}, 2^{k-1} - 1]$
Output: An arithmetic sharing modulo 3 $(w_1, \ldots, w_n)$ of $(x \bmod 3)$.

1: $y_1, \ldots, y_n \leftarrow \mathsf{AtoB}_{2^k}(x_1, \ldots, x_n)$
2: **for** $j = 0$ to $k - 1$ **do**
3: Let $y_i^{(j)}$ be the j-th bit of y_i for $1 \le i \le n$
4: $z_1^{(j)}, \ldots, z_n^{(j)} \leftarrow \mathsf{BtoA}_3(y_1^{(j)}, \ldots, y_n^{(j)})$
5: **end for**
6: **for** $i = 1$ to n **do**
7: $w_i \leftarrow \sum_{j=0}^{k-2} 2^j z_i^{(j)} - 2^{k-1} z_i^{(k-1)} \bmod 3$
8: **end for**
9: **return** $w_1, \ldots, w_n$

Security. The following theorem shows that the $\mathsf{Mod3Red}$ algorithm achieves the $t - \mathsf{SNI}$ security notion.

Theorem 1 ($t - \mathsf{SNI}$ security of $\mathsf{Mod3Red}$). *For any subset $O \subset [1, n]$ and any t_1 intermediate variables with $|O| + t_1 \le t$, the output variables $w_{|O}$ and the t_1 intermediate variables can be perfectly simulated from the input variables $x_{|I}$, with $|I| \le t_1$.*

Proof. The $t - \mathsf{SNI}$ property of the part from lines 2 to 9 follows from the $t - \mathsf{SNI}$ of each of the k independent BtoA_3 conversions. Namely the corresponding output shares $z_i^{(j)}$ are combined independently for each share index $1 \le i \le n$. Therefore we can use the same output subset O for each intermediate output shares $(z_i^{(j)})_{1 \le i \le n}$ for $0 \le j < k$. The $t - \mathsf{SNI}$ property of the complete algorithm follows from composition of two SNI gadgets. □

Complexity. We assume that a group operation as well as randomness generation takes unit time. The complexity of Algorithm 7 is therefore:

$$\begin{aligned} T_{\mathsf{Mod3Red}}(k,n) &= T_{\mathsf{AtoB}}(k,n) + k \cdot T_{\mathsf{BtoA}_3}(n) + 2 \cdot k \cdot n + 1 \\ &= \mathcal{O}(k \cdot n^2) \end{aligned}$$

4.2 Key generation: masked generation of $g \leftarrow \mathcal{L}_g$

In this section, we explain how to generate an arithmetically masked $g \leftarrow \mathcal{L}_g$, which corresponds to Line 1 of the KeyGen algorithm (Alg. 1). We consider only the HPS version, for which $\mathcal{L}_g = \mathcal{T}(q/8-2)$, see Table 1. We will consider the HRSS version in Section 7.2. Obviously, we cannot simply generate an unmasked $g \leftarrow \mathcal{L}_g$ and later arithmetically mask it with n shares, as the attacker could directly probe the unmasked g. Therefore, the key generation algorithm must be masked with n shares from the beginning.

Recall that $\mathcal{T}(q/8-2)$ is the set of ternary polynomials of degree at most $\ell-2$ containing exactly $q/16-1$ coefficients equal to 1, and $q/16-1$ coefficients equal to -1. In the NIST submission [CDH+19], the authors apply a random permutation to the coefficients of an initially fixed polynomial g_I with its first $q/16-1$ coefficients equal to 1, its $q/16-1$ following coefficients equal to -1, and its remaining coefficients equal to 0. Actually, the applied permutation is not perfectly random. Namely, in the corresponding FixedType algorithm from [CDH+19], given a $30(\ell-1)$-bit seed, the permutation is obtained by concatenating to each coefficient a 30-bit prefix, then sorting the list of 32-bit entries, and eventually discarding the 30-bit prefix to keep the permuted coefficients. Obviously, such procedure would be quite challenging to mask directly.

Alternatively, we use the following simple approach, which also provides a perfectly random permutation. We start with the initial polynomial $g = g_I$ as previously, and we encode g over $n = t+1$ shares with arithmetic masking modulo q, for security against t probes. We then repeat the following procedure $n = t+1$ times: we randomly permute the $\ell-1$ coefficients of g by generating an independent random permutation π; for this, we actually apply π on each share of g; we then perform a linear mask refreshing modulo q of each coefficients of g. Eventually, we output the arithmetically masked polynomial g modulo q. We describe the algorithm below. We denote by $\mathcal{P}_{\ell-1}$ the set of permutation of $\{0,\dots,\ell-2\}$. We assume that we have an efficient algorithm for generating a permutation $\pi \leftarrow \mathcal{P}_{\ell-1}$ uniformly at random. We recall the LinearRefresh algorithm in Appendix A.5, applied on the quotient ring $S/q = \mathbb{Z}_q[X]/\Phi_\ell$.

Algorithm 8 SecSampleT(d)

Output: $(g_1,\dots,g_n)$, an arithmetic sharing modulo q of $g \in \mathcal{T}(d)$

```
g_1, ..., g_n ← ((1 + ··· + X^{d/2−1} − X^{d/2} − ··· − X^{d−1}), 0 ..., 0)
for j = 1 to n do
    π ← P_{ℓ−1}
    for i = 1 to n do g_i ← π(g_i)
    g_1, ..., g_n ← LinearRefresh_{S/q}(g_1, ..., g_n)
end for
return (g_1, ..., g_n)
```

Security. The above algorithm is secure against an adversary with at most $t = n-1$ probes, because by definition, at least one of the n permutations and subsequent linear mask refreshing has not been probed, after which the adversary's probes can be perfectly simulated without knowing the secret key. This is the same security argument as for proving the security of the table recomputation countermeasure [Cor14]. Formally, the following theorem proves the security of the above algorithm. For a key generation algorithm, there are no inputs, so we need to prove that for any generated secret-key g, any $t < n$ probe can be perfectly simulated without knowing g.

Theorem 2 (t-probing security of $\mathsf{SecSampleT(d)}$)**.** *For any fixed secret-key $g = g_1+\cdots+g_n \pmod q$ output by* $\mathsf{SecSampleT(d)}$*, any set of $t_1 < n$ intermediate variables can be perfectly simulated without knowing g.*

Proof. We consider any fixed secret $g \in \mathcal{T}(d)$, and we consider a secret $\pi \leftarrow \mathcal{P}_{\ell-1}$ such that $g = \pi(g_I)$, where $g_I = 1 + \cdots + X^{d/2-1} - X^{d/2} - \cdots - X^{d-1}$ is the initial polynomial.

We denote by Part_j for $1 \le j \le n$ the execution steps of the algorithm during the for loop from Line 2 to Line 6. Since there are $t_1 < n$ probed variables, at least one execution of the for loop has not been probed. Let $j^\star$ be the corresponding index, such that $\mathsf{Part}_{j^\star}$ has not been probed.

We split the probed variables into 2 sets: $S^{<j^\star}$ and $S^{>j^\star}$, which correspond to the variables probed during execution of Part_j for $j < j^\star$ and $j > j^\star$ respectively. The variables from $S^{j<j^\star}$ can be perfectly simulated without the knowledge of g. Indeed, for each index $j < j^\star$, it suffices to draw $\pi_j \leftarrow \mathcal{P}_{\ell-1}$ uniformly at random, and simulate all variables from the initial sharing of g_I at Step 1 and π_j.

In order to simulate the variables from $S^{>j^\star}$, we define a set of indexes I such that $i \in I$ iff a variable g_i has been probed. By construction we have $|I| \le t_1 < n$. Since $\mathsf{Part}_{j^\star}$ has not been probed, the corresponding $\mathsf{LinearRefresh}$ gadget has not been probed, hence any subset of at most $n-1$ output shares is uniformly and independently distributed; hence the corresponding outputs $g_{|I}$ can be perfectly simulated. One can then propagate the simulation for the Part_j processes for $j > j^\star$, and simulate any variable from the set $S^{>j^\star}$ from such $g_{|I}$; as previously we generate the permutations π_j for $j > j^\star$ uniformly at random in $\mathcal{P}_{\ell-1}$.

Finally, for consistency we must have $\pi = \pi_n \circ \cdots \circ \pi_{j^\star} \circ \cdots \circ \pi_1$, which is possible by fixing the permutation $\pi_{j^\star}$ satisfying this equation. The knowledge of $\pi_{j^\star}$ is not required for the simulation, since by assumption $\mathsf{Part}_{j^\star}$ has not been probed. Hence the simulation can be performed without the knowledge of π and the output secret-key g. □

Complexity. The time complexity of the algorithm is

$$\begin{aligned} T_{\mathsf{SecSampleT}}(\ell, n) &= n \cdot (\ell - 1 + n \cdot \ell + T_{\mathsf{LinearRefresh}}(n)) \\ &= \mathcal{O}(n^2 \cdot \ell) \end{aligned}$$

Remark 1. Note that our security model assumes that the adversary can only probe at most $n-1$ of the n permutations, so in the security proof at least one permutation can be treated as a black-box. However, for security against real side-channel leakages, it may be difficult to implement a permutation so that this assumption is satisfied in practice. More precisely, it may be possible to perform a template attack against the permutations, so that using a single trace, the adversary could recover all n permutations and eventually the secret-key. We refer to [KAA21] for an example of such attack.

4.3 Key generation: high-order computation of $1/f$ modulo q

In this section, we show how to high-order compute the secret $f_q = (1/f) \bmod (q, \Phi_\ell)$ at Step 2 of KeyGen (Alg. 1). We have that f is invertible in $\mathbb{Z}[X]/(q, \Phi_\ell)$ iff f is invertible in $\mathbb{Z}[X]/(2, \Phi_\ell)$. Therefore, we first recall how to compute inverses in $S/2 = \mathbb{Z}_2[X]/\Phi_\ell$.

Computing inverse over $S/2$. Since $\Phi_\ell(x)$ is irreducible modulo 2, the multiplicative group $S/2 = \mathbb{Z}[X]/(2, \Phi_\ell)$ has order $2^{\ell-1} - 1$. Therefore, we can first compute the inversion of f in $\mathbb{Z}[X]/(2,\Phi_\ell)$, using a sequence of squares and multiplies as in [IT88], and then lift the result modulo q. Namely, such exponentiation approach is much easier to mask than the extended-gcd approach. More precisely, we must compute:

$$f^{-1} = f^{2^{\ell-1}-2} = f^{2\cdot(2^{\ell-2}-1)} \bmod (2,\Phi_\ell) \tag{5}$$

To compute this exponentiation, we use the identity $2^{a+b}-1 = 2^a\cdot(2^b-1)+(2^a-1)$, which gives:

$$f^{2^{a+b}-1} = \left(f^{2^b-1}\right)^{2^a}\cdot f^{2^a-1} \bmod (2,\Phi_\ell) \tag{6}$$

where the exponentiation by 2^a is a linear operation. In particular, we obtain:

$$f^{2^{2b}-1} = \left(f^{2^b-1}\right)^{2^b}\cdot f^{2^b-1} \bmod (2,\Phi_\ell), \qquad f^{2^{b+1}-1} = \left(f^{2^b-1}\right)^{2}\cdot f \bmod (2,\Phi_\ell)$$

which implies that we can perform the equivalent of a square-and-multiply. We provide the corresponding FastExpo algorithm below, with the proof of correctness (Theorem 3) in Appendix B.1.

Algorithm 9 FastExpo(x, m)

Input: An integer $m = (m_{k-1},\ldots,m_0)_2$ and an element $x \in \mathbb{Z}_2[X]/\Phi_\ell$
Output: x^{2^m-1} in $\mathbb{Z}_2[X]/\Phi_\ell$

$y \leftarrow 1$
for $i = k-1$ to 0 **do**
  $m' \leftarrow m \gg (i+1)$
  $y \leftarrow y \times y^{2^{m'}}$
  if $m_i = 1$ **then** $y \leftarrow y^2 \times x$
end for
return y

Theorem 3 (Correctness)**.** *Given as input $x \in \mathbb{Z}_2[X]/\Phi_\ell$, Algorithm 9 outputs x^{2^m-1} in $\lfloor \log_2 m\rfloor + \mathsf{H}_w(m) - 1 \le 2\lfloor \log_2(m)\rfloor$ non-linear multiplications, where $\mathsf{H}_w(m)$ is the Hamming weight of m.*

Computing inverse over $S/q = \mathbb{Z}_q[X]/\Phi_\ell$. We now recall how to compute inverses over $S/q = \mathbb{Z}_q[X]/\Phi_\ell$. For this we recall the unmasked SqInverse algorithm from [CDH+19], which lifts the inverse modulo 2 into an inverse modulo 2^{2^i} at each step i of the while loop, until $2^{2^i} \ge q$. We provide the proof of correctness in Appendix B.2.

Algorithm 10 SqInverse(a)

Input: An invertible polynomial $a \in S/q$
Output: A polynomial v such that $a \cdot v = 1 \bmod (q,\Phi_\ell)$

$v \leftarrow$ FastExpo$(a \bmod 2, \ell-2)$
$v \leftarrow v^2$
$t \leftarrow 1$
while $t < \log_2 q$ **do**
  $v \leftarrow v(2 - a\cdot v) \bmod (q,\Phi_\ell)$
  $t \leftarrow 2t$
end while
return v

Theorem 4 (Correctness)**.** *Algorithm* SqInverse *is correct.*

High-order masking. The two previous algorithms are easy to mask. Namely, for the FastExpo algorithm, it suffices to high-order mask the polynomial multiplications at lines 4 and 5. This can be done via a SecMult algorithm, as a straightforward extension of the And gadget from [ISW03]. We provide in Appendix B.3 the high-order masking of the FastExpo algorithm, called SecFastExpo. Similarly, we provide in Appendix B.4 an algorithmic description of the high-order masked version of Algorithm 10 above, called SecSqInverse. Note that after Line 2 of Algorithm 10, the polynomial v must be considered modulo q instead of modulo 2, so we consider each share of v as a share modulo q. The final complexity of our polynomial inversion algorithm in $S/q = \mathbb{Z}_q[X]/\Phi_\ell$ is $\mathcal{O}(n^2 \cdot (\log \ell + \log \log q))$ operations in S/q. We provide the proof of the following theorem in Appendix B.4.

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

Addition chains. More generally, to compute the exponentiation given by (5), from (6) it suffices to provide an addition chain for the integer $\ell - 2$. The number of additions in the chain gives the number of multiplications in $\mathbb{Z}[X]/(2, \Phi_\ell)$. From the square-and-multiply algorithm above, there always exists an addition chain for $m = \ell - 2$ with $\lfloor \log_2 m \rfloor + \mathsf{H}_w(m) - 1 \leq 2\lfloor \log_2(m) \rfloor$ additions. However, one can often find better addition chains. For example, in [HRSS17], the authors compute the inversion in $\mathbb{F}_{2^{700}}$ with 12 multiplications only (instead of 15 with the square-and-multiply). We refer to Appendix B.6 for more details.

5 The polynomial inversion algorithm from [KLRBG22]

Recently, the authors of [KLRBG22] described a high-order masked algorithm to perform the polynomial inversion in the key generation of NTRU, based on a conversion from arithmetic to multiplicative masking. The authors claimed that their high-order conversion algorithm can achieve arbitrary-order security, but without a security proof. Below, we show that their algorithm is actually insecure: we exhibit a 3-rd order attack for any number of shares n in the countermeasure. We then describe a simple reparation with a proof of security, and we eventually provide a comparison between our high-order inversion algorithm from Section 4.3 and the repaired algorithm.

5.1 Our third-order attack

Let $\mathcal{R}$ be a ring. The technique used in [KLRBG22] to high-order compute the inverse of an element $a \in \mathcal{R}^\star$ is to use a multiplicative masking $a = \prod_{i=1}^n m_i$ with invertible elements $m_i \in \mathcal{R}^\star$, so that the inversion in $\mathcal{R}^\star$ becomes a linear operation in the number n of masks (instead of quadratic for additive masking):

$$a^{-1} = \prod_{i=1}^{n} m_i^{-1}$$

We recall in Algorithm 11 below the arithmetic to multiplicative masking conversion algorithm from [KLRBG22, Alg. 4].

Our attack. We describe a 3-rd order attack that works for any number of shares n. We probe the initial value a_1, the value a_1' of the variable a_1 for the last index $i = 2$ after Line

Algorithm 11 Additive to multiplicative conversion (A2M)

Input: An arithmetic masking $a = a_1 + \cdots + a_n \in \mathcal{R}$
Output: A multiplicative masking $a = \prod_{i=1}^{n} m_i \in \mathcal{R}$

1: **for** $i = n$ downto 2 **do**
2: $\quad r_i \leftarrow \mathcal{R}^\star$
3: $\quad$ **for** $j = 1$ to i **do**
4: $\quad\quad a_j \leftarrow r_i \cdot a_j$
5: $\quad$ **end for**
6: $\quad m_i \leftarrow r_i^{-1}$ $\qquad \triangleright\ a = \left(\sum_{j=1}^{i} a_j\right) \prod_{j=i}^{n} m_j$
7: $\quad a_{i-1} \leftarrow a_{i-1} + a_i$
8: **end for**
9: $m_1 \leftarrow a_1$
10: **return** $m_1, \ldots, m_n$

5, and the output variable m_1. Since for each $n \geq i \geq 2$ the random r_i is multiplicatively accumulated on the variable a_1, we obtain:

$$a_1' = a_1 \cdot \prod_{i=2}^{n} r_i = a_1 \cdot \prod_{i=2}^{n} m_i^{-1}$$

which gives:

$$a = \prod_{i=1}^{n} m_i = m_1 \cdot \prod_{i=2}^{n} m_i = m_1 \cdot a_1 \cdot (a_1')^{-1}$$

which shows that the secret value a can always be recovered from the 3 probes a_1, a_1' and m_1. This shows that for any number of shares n, the countermeasure can provide at most second-order security.

In [KLRBG22, Alg. 6] the authors also described an optimization of their algorithm, which consists in converting the additive shares $a = a_1 + \cdots + a_n$ into multiplicative shares of the inverse of a, namely $a^{-1} = m_1 \times \cdots \times m_n$, using a single inversion instead of $n-1$. Our 3-rd order attack also applies against this variant. In the following, we focus on this variant since it is more efficient (as it requires a single inversion in $\mathcal{R}^\star$ instead of $n-1$ inversions). More precisely, we provide a reparation of this later algorithm, with a proof of security in the ISW probing model.

5.2 Repaired polynomial inversion algorithm

Additive to multiplicative conversion. In this section, we describe the repaired high-order polynomial inversion algorithm, starting from the additive to multiplicative conversion algorithm in [KLRBG22, Alg. 6], which requires a single polynomial inversion only. In order to repair such algorithm, it suffices to add a mask refreshing at each iteration of the for loop, and to delay the shares recombination to the end of algorithm. We provide the pseudo-code of the $\mathsf{A2M_{INV}}$ algorithm below; we refer to Appendix A.5 for the LinearRefresh algorithm. Such corrected version is actually similar to the zero-test algorithm in [CGMZ21, Algorithm 3], which is also based on an additive to multiplicative masking conversion. The time complexity of the modified algorithm is

$$T_{\mathsf{A2M}}(n) = n \cdot (1 + n + 3n - 3) + n + T_{\mathsf{inv}}(\mathcal{R}^\star) \sim 4n^2$$

Theorem 6 ($t-\mathsf{SNI}$ security of $\mathsf{A2M_{INV}}$ conversion)**.** *For any subset $O \subset [1, n]$ and any t_1 intermediate variables with $t_1 + |O| \leq t$, the output variables $m_{|O}$ and the t_1 intermediate variables can be perfectly simulated from input variables $a_{|I}$, with $|I| \leq t_1$*

Algorithm 12 Additive to multiplicative conversion ($\mathsf{A2M_{INV}}$)

Input: $a = a_1 + \cdots + a_n$
Output: $a^{-1} = m_1 \cdot \cdots \cdot m_n$

1: **for** $i = 1$ to n **do**
2: $\quad r_i \leftarrow \mathcal{R}^\star$
3: $\quad$ **for** $j = 1$ **to** n **do** $a_j \leftarrow r_i \cdot a_j$
4: $\quad a_1, \ldots, a_n \leftarrow \mathsf{LinearRefresh}_{\mathcal{R}}(a_1, \ldots, a_n)$
5: $\quad m_i \leftarrow r_i$ $\quad \triangleright\ a = \left(\sum_{j=1}^{n} a_j\right) \prod_{j=1}^{i} m_j^{-1}$
6: **end for**
7: $m_1 \leftarrow m_1 \cdot (\sum_{j=1}^{n} a_j)^{-1}$ $\quad \triangleright\ a^{-1} = m_1 \cdot m_2 \cdot \cdots \cdot m_n$
8: **return** $m_1, \ldots, m_n$

Proof. We denote by $\mathtt{Part}_i$ for $1 \leq i \leq n$ the steps of the algorithm from Line 1 to Line 6 in the `For` loop with index i, and by $a_j^{(i)}$ the value of the share a_j at the end of $\mathtt{Part}_i$. Let $P = \{i \mid \mathtt{Part}_i$ has been probed or $i \in \mathcal{O}\}$. From $t_1 + |O| < n$ we deduce $P \subsetneq [1, n]$ and therefore there exists $i^\star$ such that $\mathtt{Part}_{i^\star}$ has not been probed and $i^\star \notin \mathcal{O}$. We construct a subset $I \subset [1, n]$ of input indexes for the simulation. We start with an empty I and for each probed variable a_j we add j to the set. By construction we must have $|I| \leq t_1$.

Every probed variable in $\mathtt{Part}_i$ for $i < i^\star$ can be perfectly simulated from $a_{|I}$. It remains to simulate the variables probed at $\mathtt{Part}_i$ for $i > i^\star$. Since by assumption $m_{i^\star}$ and $r_{i^\star}$ have not been probed and $i^\star \notin \mathcal{O}$, the random $r_{i^\star}$ acts as a one-time pad for the value $a^{(i^\star)} = a_1^{(i^\star)} + \cdots + a_n^{(i^\star)}$. Moreover we note that $a^{(i^\star)} = a \cdot m_1 \cdot \cdots \cdot m_{i^\star}$ is invertible as the product of invertible elements. Therefore, $a^{(i^\star)}$ is uniformly distributed in $\mathcal{R}^\star$. Since $\mathtt{Part}_{i^\star}$ has not been probed, the corresponding $\mathsf{LinearRefresh}$ instance has not been probed. We can therefore perfectly simulate all shares $a_j^{(i^\star)}$ at the end of $\mathtt{Part}_{i^\star}$ with fresh random values whose sum is invertible. Such simulation can subsequently be propagated to all a_j variables until the end of the algorithm. We therefore conclude that Algorithm 12 is $(n-1)-\mathsf{SNI}$. □

Multiplicative to additive masking conversion. In [KLRBG22], the authors also provide a multiplicative to additive masking conversion algorithm, without a security proof. In the following, we recall their algorithm, and prove that it achieves the $t-\mathsf{SNI}$ security property. We refer to the full version of this paper [CGTZ22] for the proof. The complexity of the algorithm is $T_{\mathsf{M2}A}(n) \sim 2n^2$.

Algorithm 13 Multiplicative to additive conversion ($\mathsf{M2A}$)

Input: $m = m_1 \cdot \cdots \cdot m_n \in \mathcal{R}$
Output: $m = a_1 + \cdots + a_n \in \mathcal{R}$

$a_1 \leftarrow m_1$
for $i = 1$ to $n - 1$ **do**
$\quad a_1, \ldots, a_{i+1} \leftarrow \mathsf{LinearRefresh}_{\mathcal{R}}(a_1, \ldots, a_i, 0)$
$\quad$ **for** $j = 1$ to $i + 1$ **do** $a_j \leftarrow a_j \cdot m_{i+1}$
end for
return $a_1, \ldots, a_n$

Theorem 7 ($t - \mathsf{NI}$ security of $\mathsf{M2A}$ conversion)**.** *Any set of t probed variables can be perfectly simulated from the input variables $m_{|I}$, with $|I| \leq t$*

High-order polynomial inversion. Finally, we describe the full SNI-secure inversion algorithm based on mask conversion. The algorithm achieves the $(n-1)-\mathsf{SNI}$ security property, based on the composition of a $(n-1)-\mathsf{SNI}$ and a $(n-1)-\mathsf{NI}$ gadget. The complexity of high-order polynomial inversion in $S/q = \mathbb{Z}_q[X]/\Phi_\ell$ is $\mathcal{O}(n^2 + \log \ell)$ operations in S/q.

Algorithm 14 Inversion based on multiplicative masking $\mathsf{INV}_{\mathsf{Mul}}$

Input: $a = a_1 + \cdots a_n \in \mathcal{R}^\star$
Output: $a^{-1} = b_1 + \cdots + b_n$

$m_1, \ldots, m_n \leftarrow \mathsf{A2M}_{\mathsf{INV}}(a_1, \ldots, a_n)$
$b_1, \ldots, b_n \leftarrow \mathsf{M2A}(m_1, \ldots, m_n)$
return $b_1, \ldots, b_n$

Theorem 8 ($t-\mathsf{SNI}$ security of $\mathsf{INV}_{\mathsf{Mul}}$)**.** *For any subset $O \subset [1, n]$ and any t_1 intermediate variables with $t_1 + |O| \leq t$, the output variables $b_{|O}$ and the t_1 intermediate variables can be perfectly simulated from input variables $a_{|I}$, with $|I| \leq t_1$.*

5.3 Comparison

The repaired polynomial inversion algorithm from [KLRBG22] is asymptotically faster than our algorithm from Section 4.3, since for inversion in $S/q = \mathbb{Z}_q[X]/\Phi_\ell$, its complexity is $\mathcal{O}(n^2 + \log \ell)$ operations in S/q, instead of $\mathcal{O}(n^2 \cdot (\log \ell + \log \log q))$. This is confirmed experimentally in tables 3 and 4 below, in which we compare the cycle count and randomness consumption between the two polynomial inversion algorithms.

Table 3: Comparison for the inversion in S/q between multiplicative masking and our technique for a naive and an optimized implementation of the polynomial multiplication, in thousands of cycles.

	Security order t						
	1	2	3	4	5	6	7
SecSqInverse	18760	41912	115196	213702	332595	484385	644636
$\mathsf{INV}_{\mathsf{Mul}}$	4187	11264	21558	30964	42155	56451	71288
SecSqInverse_AVX2	743	1192	2044	2947	4115	5611	7525
$\mathsf{INV}_{\mathsf{Mul}}$_AVX2	87	141	232	357	522	700	927

Table 4: Randomness usage comparison for the inversion in S/q between multiplicative masking and our technique, in thousands of calls to the RNG outputting 32 bits of randomness.

	Security order t						
	1	2	3	4	5	6	7
SecSqInverse	19	50	94	151	221	303	398
$\mathsf{INV}_{\mathsf{Mul}}$	3	8	14	23	34	46	61

6 High-order masking of `NTRU` decryption

In the previous sections, we have considered the masking of some specific components of `NTRU`. In this section, we consider the full high-order masking of the `NTRU` `IND-CCA` decryption, more precisely the Decapsulate algorithm (Alg. 6).

We first recall the `NTRU` Decrypt and Decapsulate algorithms, already described in Section 3. The Decrypt algorithm takes as input the ciphertext c and returns (r, m) if the ciphertext c is well formed ($fail = 0$), otherwise it returns $fail = 1$. If the ciphertext is well formed, the Decapsulate algorithm returns the session key $k_1 = H_1(r, m)$, otherwise it returns the dummy key k_2.

Algorithm 3 Decrypt$((f, f_p, h_q), c)$

1: **if** $c \neq 0 \bmod (q, \Phi_1)$ **return** $(0, 0, 1)$
2: $a \leftarrow (c \cdot f) \bmod (q, \Phi_1\Phi_\ell)$
3: $m \leftarrow (a \cdot f_p) \bmod (3, \Phi_\ell)$
4: $r \leftarrow ((c - m) \cdot h_q) \bmod (q, \Phi_\ell)$
5: **if** $(r, m) \in (\mathcal{L}_r, \mathcal{L}_m)$ **return** $(r, m, 0)$
6: **else return** $(0, 0, 1)$

Algorithm 6 Decapsulate$((f, f_p, h_q, s), c)$

1: $(r, m, fail) \leftarrow$ Decrypt$((f, f_p, h_q), c)$
2: $k_1 \leftarrow H_1(r, m)$
3: $k_2 \leftarrow H_2(s, c)$
4: **if** $fail = 0$ **return** k_1
5: **else return** k_2

We summarize below the high-order masking of the Decrypt and Decapsulate operations.

1. At Step 1 of Decrypt, the input ciphertext is unmasked, so we can perform the test $c \neq 0 \bmod (q, \Phi_1)$ in clear.

2. At Step 2 of Decrypt, by assumption the secret-key f is arithmetically masked modulo q with n shares, so we obtain a masked polynomial a modulo q, by multiplying each share of f by c, as explained in Section 4.1.

3. At Step 3 of Decrypt, we must convert the masked polynomial $a = c \cdot f = 3 \cdot g \cdot r + m \cdot f \pmod{(q, \Phi_1\Phi_\ell)}$ into a masked polynomial $\tilde{a}$ modulo 3, so that the term $3 \cdot g \cdot r$ is removed by reduction modulo 3. This has been described in Section 4.1. After high-order multiplication by f_p, which is arithmetically masked modulo 3, we eventually obtain the masked message m modulo 3.

4. At Step 4 of Decrypt, we must first convert m from arithmetic masking modulo 3 to masking modulo q. See Appendix A.2 for a description of the technique. We can then obtain an arithmetic masking of r modulo q.

5. At Step 5 of Decrypt, we must test membership $r \in \mathcal{L}_r = \mathcal{T}$ and $m \in \mathcal{L}_m$ from masked r and m. We describe the corresponding high-order algorithms in sections 6.1 and 6.2 below. The bit $fail$ can be computed in the clear.

6. At Step 1 of Decapsulate, we obtain masked polynomials m and r, modulo q. For hashing (r, m) at Step 2 in Decapsulate, we must high-order mask the packS3 algorithm from [CDH+19], which is applied to (r, m) before hashing, with a Boolean masked output; see Section 6.3. We then high-order compute the hash function H_1 over Boolean shares, and the session-key k_1 is eventually returned with Boolean shares. The same procedure is applied for H_2 if $fail = 1$.

6.1 Testing membership $r \in \mathcal{L}_r = \mathcal{T}$

The membership test $r \in \mathcal{L}_r = \mathcal{T}$ is used at Step 5 of Decrypt. Recall that $\mathcal{T}$ is the set of non-zero ternary polynomials of degree at most $\ell - 2$. We actually test if $r \in \mathcal{T} \cup \{0\}$, which means that we consider (r, m) with $r = 0$ as a legitimate plaintext in the DPKE scheme. We consider the $\ell - 1$ coefficients $r^{(j)}$ of r, where each coefficient is arithmetically masked modulo q with n shares. To test if $r \in \mathcal{T} \cup \{0\}$, we must check that each of the $\ell - 1$ coefficients $r^{(j)}$ is in $\{-1, 0, 1\}$. More precisely, we must high-order compute the bit:

$$b = \bigwedge_{j=0}^{\ell-2} \left(r^{(j)} \stackrel{?}{=} -1\right) \vee \left(r^{(j)} \stackrel{?}{=} 0\right) \vee \left(r^{(j)} \stackrel{?}{=} 1\right)$$

which we can rewrite as:

$$b = \bigwedge_{j=0}^{\ell-2} \left(r^{(j)} \oplus (-1) \stackrel{?}{=} 0\right) \vee \left(r^{(j)} \stackrel{?}{=} 0\right) \vee \left(r^{(j)} \oplus 1 \stackrel{?}{=} 0\right) \tag{7}$$

In order to high-order compute (7), we first convert each coefficient $r^{(j)}$ from arithmetic to Boolean masking (see Appendix A.1). Secondly, we xor the first share with -1, 0 and 1 modulo q. Thirdly, we perform 3 zero-tests on Boolean shares to check whether the coefficient equals -1, 0 or 1 (see Appendix A.3). We then perform a secure Or between the 3 resulting tests, using $x \vee y = \overline{\overline{x} \wedge \overline{y}}$, with the same secure And gadget as in [ISW03]. Eventually, we obtain a Boolean sharing of the bit b. Since we must perform an arithmetic modulo q to Boolean conversion for each of the ℓ coefficients, the complexity is $\mathcal{O}(\ell \cdot \log(q) \cdot n^2)$.

6.2 Testing membership $m \in \mathcal{L}_m$

The membership test $r \in \mathcal{L}_m$ is used at Step 5 of Decrypt. In the HRSS version, we have $\mathcal{L}_m = \mathcal{T}$ (see Table 1), and since the coefficients of m are ternary by definition (as they are obtained modulo 3), we do not need to perform any additional test. For the HPS version, we have $\mathcal{L}_m = \mathcal{T}(q/8-2)$, so we need to check that m has $q/16-1$ coefficients equals to 1 and $q/16-1$ coefficients equals to -1. To do so we first check if the sum of the coefficients of m is zero, and we then test if the sum of squared coefficients of m is $q/8-2$. More precisely, given the $\ell-1$ coefficients $(m^{(0)}, \ldots, m^{(\ell-2)})$ of m, we high-order compute the bit:

$$b = \left(\sum_{j=0}^{\ell-2} m^{(j)} \bmod q \stackrel{?}{=} 0\right) \wedge \left(\sum_{j=0}^{\ell-2} (m^{(j)})^2 - (q/8-2) \bmod q \stackrel{?}{=} 0\right)$$

For this, we need to perform two zero-tests on arithmetic sharing modulo q, starting from an arithmetic masking modulo q of the coefficients of m (which is also required for the high-order computation of r at Step 4 of Decrypt); see Appendix A.4. The complexity is $\mathcal{O}((\log(q) + \ell) \cdot n^2)$.

Note that for the testing of $m \in \mathcal{L}_m$ and $r \in \mathcal{L}_r$, the adversary should not learn whether $m \in \mathcal{L}_m$ and $r \in \mathcal{L}_r$ separately, so we must keep the result of both tests in masked form before returning the result of the And of the two tests. However this final result ($fail = 0$ or $fail = 1$) is not sensitive and can be computed in the clear. The total complexity is $\mathcal{O}(\ell \cdot \log(q) \cdot n^2)$.

6.3 Packing ternary polynomials

In the NIST submission of NTRU [CDH+19], the authors describe the PackS3 algorithm for converting ternary polynomials into a sequence of bytes. In particular, the PackS3 algorithm is used for computing the hash $k_1 = H_1(r, m)$ at Step 2 of Decapsulate.

More precisely, given as input a vector v of 5 ternary coefficients $v = (v_0, \ldots, v_4) \in \{0, 1, 2\}^5$, the packS3 algorithm interprets the vector v as an integer $0 \leq x < 243$ in base 3:

$$x = \sum_{j=0}^{4} 3^j \cdot v_j \tag{8}$$

which is then converted into a 8-bit string. The above procedure is applied sequentially on chunks of five coefficients of the polynomial until no coefficient is left.

When the polynomials r and m are arithmetically masked modulo 3, the above coefficients v_j's are also masked modulo 3. Therefore, we first perform an arithmetic modulo 3 to arithmetic modulo 256 conversion of each coefficient v_j (we refer to Appendix A.2 for a full description of the conversion algorithm):

$$\begin{aligned} v_j &= v_{j,1} + \cdots + v_{j,n} \pmod 3 \\ &= w_{j,1} + \cdots + w_{j,n} \pmod{256} \end{aligned} \tag{9}$$

Combining (8) and (9), we obtain an arithmetic masking of x modulo 256:

$$x = \sum_{j=0}^{4} 3^j \cdot \sum_{i=1}^{n} w_{j,i} = \sum_{i=1}^{n} \left(\sum_{j=0}^{4} 3^j \cdot w_{j,i} \right) \pmod{256}$$

Eventually we perform an arithmetic to Boolean conversion of x. The final complexity is $\mathcal{O}(n^2)$ for n shares.

In the algorithm above we have assumed that the polynomial r is initially masked modulo 3, while after Step 4 of the Decrypt algorithm (Alg. 3), the polynomial r is actually masked modulo q. However, we know after Line 5 that the polynomial r must be ternary. Therefore, we can use the Mod3Red algorithm from Section 4.1 to obtain an arithmetic masking modulo 3 of r. We also describe in Appendix C another method to pack ternary polynomials when they are arithmetically masked modulo q.

7 High-order masking of NTRU key generation

In this section, we consider the high-order masking of the NTRU key generation. We first recall the KeyGen algorithm, already described in Section 3.

Algorithm 1 KeyGen

1: $f \leftarrow \mathcal{L}_f$, $g \leftarrow \mathcal{L}_g$
2: $f_q \leftarrow (1/f) \bmod (q, \Phi_\ell)$
3: $h \leftarrow (3 \cdot g \cdot f_q) \bmod (q, \Phi_1\Phi_\ell)$
4: $h_q \leftarrow (1/h) \bmod (q, \Phi_\ell)$
5: $f_p \leftarrow (1/f) \bmod (3, \Phi_\ell)$
6: **return** $((f, f_p, h_q), h)$

We summarize below the high-order masking of the KeyGen algorithm:

1. At Step 1 of KeyGen, we must obtain the masked secret $f \leftarrow \mathcal{L}_f$. In the HPS version, $\mathcal{L}_f = \mathcal{T}$, which is the set of non-zero ternary polynomials. We describe the corresponding algorithm in Section 7.1. In the HRSS version, we have $\mathcal{L}_f = \mathcal{T}_+$. We describe the corresponding algorithm in Section 7.2. In both cases, we output both an arithmetic masking modulo 3 and an arithmetic masking modulo q of the polynomial f.

2. Similarly, we must generate $g \leftarrow \mathcal{L}_g$. The polynomial g must be masked modulo q. In the HPS version, we must sample $g \in \mathcal{T}(q/8 - 2)$. The procedure was already described in Section 4.3. In the HRSS version, we must sample $g \leftarrow \Phi_1 \cdot \mathcal{T}_+$, see Section 7.2.

3. At Step 2, we must mask the inversion $f_q \leftarrow (1/f) \bmod (q, \Phi_\ell)$, starting from an arithmetic masking modulo q of f. The inversion can be computed as a sequence of squares and multiplies in the finite field modulo $(2, \Phi_\ell)$, and then lifted by a sequence of multiplications to modulo (q, Φ_ℓ). This was already considered in Section 4.3.

4. At Step 3, we compute a high-order multiplication of g and f_q to obtain the public-key h, whose shares are recombined. The inversion at Step 4 is then done in the clear. Namely, h_q is part of the secret key only to fasten the recomputation of r during the CCA decryption, but h_q does not need to be secret since it can be computed from the public key h.

5. Finally, at Step 5, we must also high-order compute the inversion $f_p \leftarrow (1/f) \bmod (3, \Phi_\ell)$. This is also performed as a sequence of squares and multiplies in the finite field modulo $(3, \Phi_\ell)$, as when working modulo 2. We describe this procedure in Appendix D.

7.1 Masked generation of $f \leftarrow \mathcal{L}_f$ with $\mathcal{L}_f = \mathcal{T}$ (HPS version)

We describe the high-order masked generation of $f \leftarrow \mathcal{L}_f$ at Step 1 of KeyGen. We first consider the HPS version where $\mathcal{L}_f = \mathcal{T}$; we will consider the HRSS version in the next section. Recall that $\mathcal{T}$ is the set of non-zero ternary polynomials of degree at most $\ell - 2$. Therefore $|\mathcal{T}| = 3^{\ell-1} - 1$. For simplicity we can actually generate a random $f \in \mathcal{T} \cup \{0\}$, so that we can generate each coefficient of f in $\{-1, 0, 1\}$ independently. [2]

The high-order sampling is straightforward: we simply generate independently n polynomials f_i for $1 \le i \le n$ with random coefficients modulo 3. The polynomials f_i's will be the n arithmetic shares modulo 3 of the secret polynomial f:

$$f = \sum_{i=1}^{n} f_i \pmod 3$$

Recall that we must also obtain an arithmetic sharing modulo q of f. For this we will convert each coefficient $f^{(j)}$ of f from masking modulo 3 to modulo q. This is easily done by applying the table-based conversion algorithm from [CGMZ22], see Appendix A.2.

7.2 Masked generation of $f \leftarrow \mathcal{L}_f$ with $\mathcal{L}_f = \mathcal{T}_+$ (HRSS version)

In the HRSS version of the scheme, one must sample the polynomial f in the set $\mathcal{T}_+$, which is a subset of $\mathcal{T}$ containing solely polynomials $\sum_{i=0}^{\ell-2} v^{(i)} X^i$ such that $\sum_{i=0}^{\ell-2} v^{(i)} \cdot v^{(i+1)} \ge 0$. Elements of $\mathcal{T}_+$ are said to be non-negatively correlated; we refer to [CDH+19, Section 2.2.4] for the motivation of generating f in $\mathcal{T}_+$ rather than $\mathcal{T}$.

We first describe the unmasked version. We first randomly generate a random element $v \leftarrow \mathcal{T}$, with $v = \sum_{i=0}^{\ell-2} v^{(i)} X^i$. We then compute the correlation:

$$t = \sum_{i=0}^{\ell-2} v^{(i)} \cdot v^{(i+1)} \tag{10}$$

If $t < 0$, we flip the sign of even-indexed coefficients, so that we obtain a positive t. Indeed, letting v' be the polynomial with flipped coefficients and letting t' be its correlation, we obtain:

$$t' = \sum_{i=0}^{\ell-2} v'^{(i)} \cdot v'^{(i+1)} = \sum_{i=0}^{\ell-2} -v^{(i)} \cdot v^{(i+1)} = -t > 0$$

For the high-order masked version, we start from a high-order masked $v \leftarrow \mathcal{T}$ from the procedure of Section 7.1, with an arithmetic masking modulo q. We can high-order compute the value t in (10) using a sequence of secure multiplications and additions modulo q. The sign of t can then be retrieved by converting to Boolean masked form

[2] In [CDH+19], the polynomial f is generated by the Ternary algorithm, which samples each coefficient independently from $\{-1, 0, 1\}$, but with a slightly biased distribution.

and extracting the most significant bit. This sign bit is not sensitive, since eventually we must have $t \geq 0$. Therefore it can be unmasked, and if $t < 0$ we can flip the even-indexed coefficients over the arithmetic shares modulo q. Note that the value of t can be computed modulo q, because we must have $|t| < \ell < q/2$. The complexity is $\mathcal{O}((\log(q) + \ell) \cdot n^2)$.

Masked generation of $g \leftarrow \mathcal{L}_g = \Phi_1 \cdot \mathcal{T}_+$ (HRSS version). We proceed similarly for the generation of $g \leftarrow \mathcal{L}_g = \Phi_1 \cdot \mathcal{T}_+$, simply by generating a random element in $\mathcal{T}_+$ as above, and then multiplying by Φ_1.

8 Implementation results and concrete evaluation

8.1 Implementation results

In order to assess the practicality and scalability at high-order of our countermeasure, we have performed a proof of concept implementation in C. The source code can be found at

https://github.com/fragerar/Masked_NTRU

We have run our implementation on a laptop equipped with an Intel CPU, and also on a Cortex-M3 core mounted on an Arduino Due board. Random numbers are generated using a simple xorshift PRNG, a secure implementation should replace it by a cryptographically secure PRNG or a TRNG.

Performances on Intel CPU. We provide the running times for various security orders t in tables 5, 6, 7 and 8. More precisely, in Table 5, we display the cycle counts for the masked version of the decapsulation procedure incorporated in the reference code, across all parameters sets. The scaling seems to be quite reasonable for all versions of `NTRU`. However, this result is slightly biased by the fact that the polynomial multiplication used in the reference code of `NTRU` is not optimized. Indeed, this operation is relatively slow, and therefore the overhead incurred by our new gadgets is relatively low, since a large amount of time is spent in the polynomial multiplications.

Table 5: Cycle counts for decapsulation for all parameters of `NTRU`, in thousands of cycles, on `Intel(R) Core(TM) i7-1065G7 CPU @1.30GHz`.

	Security order t								
	0	1	2	3	4	5	6	7	8
ntruhps2048509	716	2 178	4 496	7 715	12 217	18 645	25 986	31 533	38 717
ntruhps2048677	1 074	3 406	7 582	12 537	19 658	29 395	37 610	51 538	74 899
ntruhrss701	1 219	3 777	8 329	13 887	21 526	30 259	40 560	59 834	83 818
ntruhps4096821	1 593	4 917	11 190	18 196	28 805	39 129	60 898	90 625	123 323

Similarly, we provide in tables 6 and 7 the cycle count for the key generation, using the exponentiation method from Section 4.3 and the multiplicative method from Section 5.2. We see that as in Section 5.3, the later is more efficient.

Table 6: Cycle counts for key generation (exponentiation method) for all parameters of `NTRU`, in thousands of cycles, on `Intel(R) Core(TM) i7-1065G7 CPU @ 1.30GHz`.

	Security order t			
	0	1	2	3
ntruhps2048509	3 565	33 060	73 685	130 005
ntruhps2048677	6 398	71 054	129 927	287 812
ntruhrss701	7236	71 560	138 982	269 223
ntruhps4096821	8 580	82 550	202 390	333 269

Table 7: Cycle counts for key generation (multiplicative method) for all parameters of `NTRU`, in thousands of cycles, on `Intel(R) Core(TM) i7-1065G7 CPU @ 1.30GHz`.

	Security order t			
	0	1	2	3
ntruhps2048509	3 565	10 339	14 858	22 306
ntruhps2048677	6 398	18 307	27 012	38 778
ntruhrss701	7 236	16 635	34 845	57 310
ntruhps4096821	8 580	23 322	38 745	59 819

We also provide in Table 8 the cycle counts using the AVX2 optimized version of the reference code for the ntruhps2048509 parameter set, significantly reducing the cost of polynomial multiplication. We obtain a significant speed-up for the Decapsulate, KeyGen (exponentiation method) and KeyGen' (multiplicative method) algorithms. In particular, since KeyGen and KeyGen' consist almost only in polynomial multiplications (and randomness generation), their runtime is hugely reduced by the AVX2 optimizations, which makes it competitive with the decapsulation. On the other hand, the overhead to mask the decapsulation is now way larger, since gadgets not depending on the polynomial arithmetic are taking a larger amount of the runtime.[3] We also display in Table 8 the relative performances of the gadgets. We see that the reduction modulo 3 and the ternary check are the most time consuming, because of the conversions between arithmetic and Boolean masking.

Table 8: Cycle counts for key generation, decapsulation and main gadgets for the optimized AVX2 version of ntruhps2048509, in thousands of cycles

	Security order t								
	0	1	2	3	4	5	6	7	8
Decaps	20	608	1758	3210	5479	8683	12019	15453	19505
KeyGen	89	753	1915	3599	6877	8661	12976	17684	21032
KeyGen'	89	323	581	1077	1782	2619	3520	4644	6207
sec_S3_mul	–	15	37	84	130	193	268	333	467
poly_mod3_reduce	–	249	671	1457	2313	3633	5028	6443	8879
ternary_check	–	92	420	676	1234	1786	2350	2960	3913
pack_S3	–	26	93	187	317	491	667	874	1205
check_message_space	–	47	94	198	307	471	653	869	1141
lift	–	24	56	132	219	349	501	660	893

Randomness usage We provide in Table 9 the randomness usage of the full decryption (Decapsulate) and of the key generation (KeyGen and KeyGen'); we also provide the

[3] Note that it would also be possible to write the other gadgets in AVX2 to speed them up, but the benefit is likely to be reduced compared to polynomial arithmetic which is a highly structured operation.

randomness consumption of the main gadgets. As expected, the number of calls to the RNG is growing significantly when the order increases. In general, randomness usage is strongly correlated to performances, because shares refreshing is needed at the core of most gadgets to ensure security in the probing model. The exceptions are gadgets that manipulate polynomials with small coefficients such as the masked multiplication of ternary polynomials and the key generation procedure. Indeed, they are cheap in terms of randomness since multiple coefficients can be extracted from a 32-bit integers but are still performing the expensive polynomial multiplication in the ring. Note that for the gadgets performing refreshes modulo q, a whole call to the RNG is counted for each value in $\mathbb{Z}_q$. In practice, at least two values could be extracted from the 32-bit output of the RNG, but it was not done for the sake of simplicity and to avoid potential leakage due to multiple random elements of $\mathbb{Z}_q$ depending on the same initial random value.

Table 9: Randomness usage for key generation, decapsulation and main gadgets, in thousands of calls to the RNG outputting 32 bits of randomness.

	Security order t							
	1	2	3	4	5	6	7	8
Decapsulate	52	205	419	745	1147	1621	2170	2842
KeyGen	31	84	161	264	393	552	740	960
KeyGen'	13	38	78	134	207	301	415	551
sec_S3_mul	0.042	0.129	0.258	0.428	0.641	0.897	1.189	1.543
poly_mod3_reduce	17	74	154	278	432	612	822	1080
ternary_check	15	61	122	219	337	475	633	832
pack_S3	2	10	21	38	59	84	113	149
check_message_space	3	10	22	37	57	80	108	140
lift	2	7	15	26	41	58	78	102

Embedded implementation. In addition, since masking schemes are mainly aimed at embedded devices, we have also tested our code on a Cortex-M3 core mounted on an Arduino Due board. The cycle counts on this platform for the decapsulation and the key generation of ntruhps2048509 are displayed in Table 10. We see that the scaling of the masking scheme at different orders is mostly similar to the results of tables 5 and 6. This is not surprising since the implementation is in plain C and not optimized for any particular architecture.

Table 10: Cycle counts for decapsulation and key generation of ntruhps2048509 on a Cortex-M3 CPU, in thousands of cycles

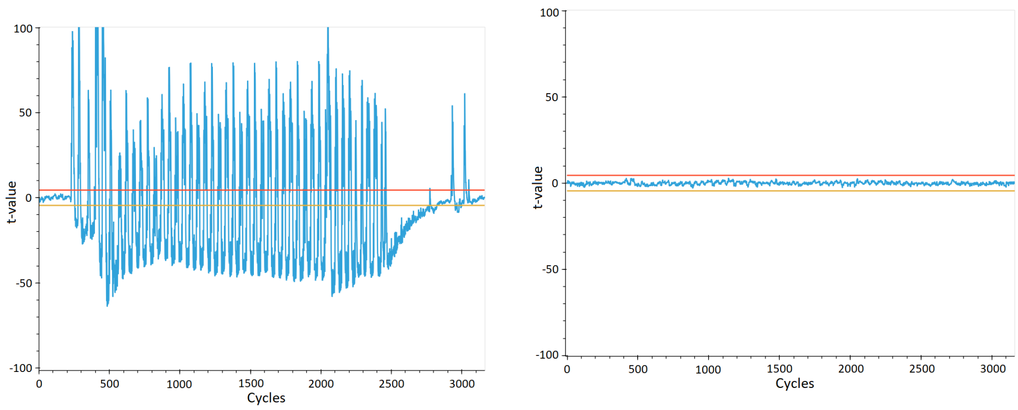
	Security order t				
	0	1	2	3	4
Decaps	10 508	32 472	70 357	117 367	182 471
KeyGen	117 348	541 752	1 152 565	1 992 624	3 051 656

8.2 Concrete leakage evaluation

Finally, we also provide some security guarantees by performing a fixed vs random t-test over 10 000 traces for one of the main gadgets, namely the reduction modulo 3 described in Section 4.1. The results can be found in Figure 1. The platform used for the experiments is a ChipWhisperer-Lite board that embeds a Cortex-M4 microcontroller (STM32F303) and a light oscilloscope.

For the leakage assessment, we have rewritten the gadget specifically at order 1 in ARM assembly, to avoid potential side-channel unsafe modifications from the compiler. We have conducted a fixed versus random t-test using the methodology described in [SM15]. The technique consists in performing the power consumption measurements while the device is executing the targeted gadget either with a fixed secret value chosen beforehand, or with a random value sampled before each measurement. This creates two sets of traces corresponding to the fixed vs the random values respectively. The t-test will then be used as a distinguisher between the two sets at each point in the power traces. If the values output by the t-test are high, it means that the statistical difference could potentially be used by the adversary to learn something about the secret key. In practice, we have used a set of 10 000 traces. For each trace, a coin was flipped to determine whether the random or the fixed secret value should be used.

We see in Figure 1 that when the RNG is switched off with randomness set to 0 (that is, without refreshing the shares), the random and fixed inputs are distinguishable as the t-values are well above the usual threshold $|t| > 4.5$. When the random number generator is switched on, values are properly masked and the test is successful on the gadget.



(a) RNG off **(b)** RNG on

Figure 1: t-test results on a ChipWhisperer-Lite board, with 10 000 traces.

9 Conclusion

In this paper, we have described the first fully masked implementation of the `NTRU` Key Encapsulation Mechanism submitted to NIST (IND-CCA decapsulation and key generation), with a security proof in the ISW probing model. We have provided a concrete implementation on ARM Cortex-M3 architecture, showing that our implementation is reasonably efficient, and also a t-test leakage evaluation. Finally, we have described a 3-rd order attack against a high-order polynomial inversion algorithm for `NTRU` recently published in [KLRBG22], and a repaired algorithm with a security proof in the ISW probing model.

References

[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.

[BGR+21] Joppe W. Bos, Marc Gourjon, Joost Renes, Tobias Schneider, and Christine van Vredendaal. Masking Kyber: First- and higher-order implementations. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2021(4):173–214, 2021. https://eprint.iacr.org/2021/483.

[BP18] Daniel J Bernstein and Edoardo Persichetti. Towards kem unification. *Cryptology ePrint Archive*, 2018.

[CDH+19] Cong Chen, Oussama Damba, Jeffrey Hoffstein, Andreas Hülsing, Joost Rijneveld, John M. Schanck, Peter Schwabe, William Whyte, and Zhenfei Zhang. NTRU: Algorithm specifications and supporting documentation. *Brown University and Onboard security company, Wilmington USA*, 2019.

[CGMZ21] Jean-Sébastien Coron, François Gérard, Simon Montoya, and Rina Zeitoun. High-order polynomial comparison and masking lattice-based encryption. Cryptology ePrint Archive, Report 2021/1615, 2021. https://ia.cr/2021/1615.

[CGMZ22] Jean-Sébastien Coron, François Gérard, Simon Montoya, and Rina Zeitoun. High-order table-based conversion algorithms and masking lattice-based encryption. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2022(2):1–40, 2022. https://ia.cr/2021/1314.

[CGTZ22] Jean-Sebastien Coron, François Gérard, Matthias Trannoy, and Rina Zeitoun. High-order masking of ntru. Cryptology ePrint Archive, Paper 2022/1188, 2022. https://eprint.iacr.org/2022/1188. Full version of this paper.

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

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

[EMVW22] Andre Esser, Alexander May, Javier Verbel, and Weiqiang Wen. Partial key exposure attacks on bike, rainbow and ntru. Cryptology ePrint Archive, Report 2022/259, 2022. https://ia.cr/2022/259.

[FO99] Eiichiro Fujisaki and Tatsuaki Okamoto. Secure integration of asymmetric and symmetric encryption schemes. In *CRYPTO '99, 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.

[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.

[HPS98] Jeffrey Hoffstein, Jill Pipher, and Joseph H Silverman. Ntru: A ring-based public key cryptosystem. In *International algorithmic number theory symposium*, pages 267–288. Springer, 1998.

[HRSS17] Andreas Hülsing, Joost Rijneveld, John M. Schanck, and Peter Schwabe. High-speed key encapsulation from NTRU. In Wieland Fischer and Naofumi Homma, editors, *Cryptographic Hardware and Embedded Systems - CHES 2017 - 19th International Conference, Taipei, Taiwan, September 25-28, 2017, Proceedings*, volume 10529 of *Lecture Notes in Computer Science*, pages 232–252. Springer, 2017.

[ISW03] Yuval Ishai, Amit Sahai, and David A. Wagner. Private circuits: Securing hardware against probing attacks. In *CRYPTO 2003, Proceedings*, pages 463–481, 2003.

[IT88] Toshiya Itoh and Shigeo Tsujii. A fast algorithm for computing multiplicative inverses in gf(2m) using normal bases. *Information and Computation*, 78(3):171–177, 1988.

[KAA21] Emre Karabulut, Erdem Alkim, and Aydin Aysu. Single-trace side-channel attacks on ω-small polynomial sampling: With applications to ntru, NTRU prime, and CRYSTALS-DILITHIUM. In *IEEE International Symposium on Hardware Oriented Security and Trust, HOST 2021, Tysons Corner, VA, USA, December 12-15, 2021*, pages 35–45. IEEE, 2021. https://eprint.iacr.org/2022/494.

[KLRBG22] Markus Krausz, Georg Land, Jan Richter-Brockmann, and Tim Güneysu. Efficiently masking polynomial inversion at arbitrary order. Cryptology ePrint Archive, Paper 2022/707, 2022. https://eprint.iacr.org/2022/707.

[PPM17] Robert Primas, Peter Pessl, and Stefan Mangard. Single-trace side-channel attacks on masked lattice-based encryption. In *CHES 2017, Proceedings*, pages 513–533, 2017.

[REB+21] Prasanna Ravi, Martianus Frederic Ezerman, Shivam Bhasin, Anupam Chattopadhyay, and Sujoy Sinha Roy. Will you cross the threshold for me?-generic side-channel assisted chosen-ciphertext attacks on ntru-based kems. *Cryptology ePrint Archive*, 2021.

[RP10] Matthieu Rivain and Emmanuel Prouff. Provably secure higher-order masking of AES. In *CHES 2010, Proceedings*, pages 413–427, 2010.

[SM15] Tobias Schneider and Amir Moradi. Leakage assessment methodology - A clear roadmap for side-channel evaluations. In *CHES 2015. Proceedings*, volume 9293 of *Lecture Notes in Computer Science*, pages 495–513. Springer, 2015.

[SMS19] Thomas Schamberger, Oliver Mischke, and Johanna Sepulveda. Practical evaluation of masking for ntruencrypt on arm cortex-m4. In *International Workshop on Constructive Side-Channel Analysis and Secure Design*, pages 253–269. Springer, 2019.

[SPOG19] Tobias Schneider, Clara Paglialonga, Tobias Oder, and Tim Güneysu. Efficiently masking binomial sampling at arbitrary orders for lattice-based crypto. In *PKC 2019, Proceedings, Part II*, pages 534–564, 2019.

[SS13] Damien Stehlé and Ron Steinfeld. Making ntruencrypt and ntrusign as secure as standard worst-case problems over ideal lattices. Cryptology ePrint Archive, Report 2013/004, 2013. https://ia.cr/2013/004.

[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 Existing masking gadgets

In this section, we summarize the main masking gadgets used in the definition of our algorithms, with their running-time complexity and security property.

A.1 Conversion between arithmetic and Boolean masking

For the high-order masking of NTRU, we need to convert between arithmetic masking modulo 2^k and Boolean masking. Such high-order conversion was first described in [CGV14], with complexity $\mathcal{O}(n^2 \cdot k)$ for n shares and k-bit words, with the NI property, in both directions. To obtain the SNI property, it suffices to compose with a SNI mask refreshing. These conversion algorithms were later extended by [BBE+18] to arithmetic masking modulo any integer q, with complexity $\mathcal{O}(n^2 \cdot k)$ or even $\mathcal{O}(n^2 \cdot \log k)$, where $k = \log_2(q)$, still with the SNI property.

Recently, a different algorithm was described in [CGMZ22], based on randomized table-recomputation, with the same complexity $\mathcal{O}(n^2 \cdot k)$ in both directions, and satisfying the SNI property. An alternative algorithm for converting from Boolean to arithmetic masking is also described in [SPOG19], with the same property.

In summary, we can assume that we have SNI conversion algorithms denoted AtoB_q and BtoA_q, to convert between arithmetic masking modulo q and Boolean masking, with asymptotic complexity $\mathcal{O}(n^2 \cdot \log q)$ in both directions, and satisfying the SNI property.

A.2 Arithmetic modulo 3 to modulo q conversion

We describe the conversion from arithmetic masking modulo 3 to masking modulo 2^k. One could use the composition of two conversions with Boolean masking as a intermediate step, with complexity $\mathcal{O}(n^2 \cdot k)$. Alternatively, a direct approach based on table recomputation is easier and more efficient, with complexity $\mathcal{O}(n^2)$ only.

More precisely, in [CGMZ22], the authors described the high-order computation of any function $f : G \to H$ where G and H are arbitrary groups. We instantiate their generic conversion with $G = \mathbb{Z}_3$, $H = \mathbb{Z}_{2^k}$ and the injection $f : \mathbb{Z}_3 \to \mathbb{Z}_{2^k}$ that maps $0, 1, -1$ to $0, 1, (2^k - 1)$ respectively. This leads to the following algorithm below (Alg. 18), with complexity $\mathcal{O}(n^2)$. It uses a table T with 3 rows $T(0)$, $T(1)$ and $T(2)$ of n shares each. As shown in [CGMZ22], the algorithm satisfies the SNI property.

A.3 Zero-testing over Boolean shares

We consider the zero-testing of a value $x \in \{0,1\}^k$ over Boolean shares. More precisely, the algorithm takes as input a Boolean sharing of x, and returns a Boolean sharing of $b \in \{0,1\}$ such that $b = 1$ if and only if $x = 0$. Writing $x = (x^{(0)}, \ldots, x^{(k-1)})_2$ the k bits of x, we have $b = \bigwedge_{i=0}^{k-1} \overline{x^{(i)}}$. Therefore the bit b can be high-order computed by using high-order secure And gadgets, with the SNI property. We refer to [CGMZ21] for the description of such an algorithm, with complexity $T_{\mathsf{ZeroTestBool}}(k, n) = \mathcal{O}(k \cdot n^2)$.

Algorithm 18 $\mathsf{Convert}_{\mathbb{Z}_3,\mathbb{Z}_{2^k}}(x_1,\ldots,x_n)$

Input: $(x_1,\ldots,x_n)\in\mathbb{Z}_3^n$
Output: $(y_1,\ldots,y_n)\in\mathbb{Z}_{2^k}^n$ with $\sum_{i=1}^n y_i = x \pmod{2^k}$, $\sum_{i=1}^n x_i = x \pmod 3$ and $x\in\{0,1,-1\}$.

$T(0)\leftarrow(0,0,\ldots,0)$
$T(1)\leftarrow(1,0,\ldots,0)$
$T(2)\leftarrow(2^k-1,0,\ldots,0)$
for $i=1$ to $n-1$ **do**
  for $u=0$ to 2 **do**
    for $j=1$ **to** n **do** $T'(u)[j]\leftarrow T(u+x_i \bmod 3)[j]$
  end for
  for $u=0$ to 2 **do**
    $T(u)\leftarrow \mathsf{Refresh}_{\mathbb{Z}_{2^k}}(T'(u))$
  end for
end for
$y_1,\ldots,y_n\leftarrow \mathsf{Refresh}_{\mathbb{Z}_{2^k}}(T(x_n))$
return $y_1,\ldots,y_n$

A.4 Zero-testing over arithmetic shares

For the zero-testing over arithmetic shares, we refer to [CGMZ21] for the description of various techniques. A first technique consists in first applying an arithmetic to Boolean conversion and then applying the zero-testing over the Boolean shares as in the previous section. Another method for prime moduli is based on Fermat's little theorem. A third method, also for prime moduli, is based on converting from arithmetic to multiplicative masking. Eventually, we assume that we have an SNI zero-test algorithm ZeroTestArith taking as input an arithmetic sharing modulo 2^k of a value x, and returning a Boolean sharing of b such that $b=1$ if and only if $x=0$, with complexity $T_{\mathsf{ZeroTestArith}}(k,n)=\mathcal{O}(k\cdot n^2)$

A.5 Linear mask refreshing

We recall the LinearRefresh algorithm from [RP10], working in any additive group G:

Algorithm 19 LinearRefresh

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

$y_n\leftarrow x_n$
for $j=1$ to $n-1$ **do**
  $r_j\leftarrow G$
  $y_j\leftarrow x_j+r_j$
  $y_n\leftarrow y_n-r_j$
end for
return $y_1,\ldots,y_n$

B Computing inverses in S/q

B.1 Proof of Theorem 3 (correctness of exponentiation in $\mathbb{Z}_2[X]/\Phi_\ell$)

We claim Algorithm 9 is correct. Let $x \in \mathbb{Z}_2[X]/\Phi_\ell$ and $m \in \mathbb{N}$. We show by induction on $k-1 \geq i \geq 0$ that at the end of each iteration of the loop, the value y_i of the variable y satisfies $y_i = x^{2^{M_i}-1}$, where $M_i = m \gg i$. For $i = k-1$, we have $M_{k-1} = m_{k-1} = 1$, hence $y_{k-1} = x = x^{2^{M_{k-1}}-1}$ as required. We now assume the result holds at iteration i and we show that the result holds at step $i-1$. From the square step, we have $y'_i = (y_i)^{2^{M_i}} \times y_i$, and after the multiply step, we have $y_{i-1} = (y'_i)^{2^{m_{i-1}}} \times x^{m_{i-1}}$, which gives $y_{i-1} = y_i^{2^{m_{i-1}}} \times y_i^{2^{M_i+m_{i-1}}} \times x^{m_{i-1}} = (y_i^{2^{M_i}+1})^{2^{m_{i-1}}} \times x^{m_{i-1}}$. By induction hypothesis $y_i = x^{2^{M_i}-1}$, so we obtain $y_{i-1} = x^e$ with

$$
\begin{aligned}
e &= (2^{M_i}-1)\cdot(2^{M_i}+1)\cdot 2^{m_i} + m_{i-1} = (2^{2M_i}-1)\cdot 2^{m_{i-1}} + m_{i-1} \\
&= 2^{2M_i+m_{i-1}} + m_{i-1} - 2^{m_{i-1}}
\end{aligned}
$$

From $2\cdot M_i + m_{i-1} = M_{i-1}$ and $m_{i-1} - 2^{m_{i-1}} = -1$ we deduce $e = 2^{M_{i-1}} - 1$. Hence the induction step is proven. Therefore $y_0 = x^{2^{M_0}-1} = x^{2^m-1}$ and the algorithm is correct.

Moreover we need a multiplication for each square step and from each multiply step with exception of the first square step which corresponds to $1 * 1$. This lead to a number of multiplications:

$$
\lfloor \log_2(m) \rfloor + H_w(m) - 1 \leq 2\lfloor \log_2(m) \rfloor
$$

B.2 Proof of Theorem 4

We claim that Algorithm 10 is correct. Indeed, we show by induction that at the beginning of each step i of the while loop we have $t_i = 2^i$ and $v_i \cdot a = 1 \pmod{(2^{t_i}, \Phi_\ell)}$, where v_i denotes the variable v at Step i. At step $i = 0$, by definition we have $t_0 = 1$. Moreover we have $v_0 \cdot a = 1 \bmod (2, \Phi_\ell)$.

We now prove the induction step, assuming that $t_i = 2^i$ and $v_i \cdot a = 1 \pmod{(2^{t_i}, \Phi_\ell)}$ holds. First, we have $t_{i+1} = 2t_i = 2^{i+1}$. We have:

$$
\begin{aligned}
1 - a\cdot v_{i+1} &= 1 - a\cdot v_i \cdot (2 - a\cdot v_i) \pmod{(2^{2t_i}, \Phi_\ell)} \\
&= (1 - a\cdot v_i)^2 \pmod{(2^{2t_i}, \Phi_\ell)}
\end{aligned}
$$

From the induction hypothesis, we can write $1 - a\cdot v_i = P \cdot 2^{t_i} \pmod{\Phi_\ell}$ for some polynomial $P \in \mathbb{Z}[x]$, which gives:

$$
\begin{aligned}
1 - a\cdot v_{i+1} &= P^2 \cdot 2^{2t_i} \pmod{(2^{2t_i}, \Phi_\ell)} \\
&= 0 \pmod{(2^{t_{i+1}}, \Phi_\ell)}
\end{aligned}
$$

which proves the induction step, and therefore the correctness of the SqInverse algorithm.

B.3 Secure exponentiation modulo 2

We provide in Algorithm 20 the high-order masking of the FastExpo algorithm recalled in Section 4.3. We assume that we have a SecMult algorithm for high-order computing the product of two polynomials in $\mathbb{Z}_2[X]/\Phi_\ell$, with the SNI property. It can be obtained as a straightforward extension of the And gadget from [ISW03].

Algorithm 20 SecFastExpo$((x_1, \ldots, x_n), m)$

Input: An integer $m = (m_{k-1}, \ldots, m_0)_2$, and an arithmetic sharing modulo 2 of $x \in \mathbb{Z}_2[X]/\Phi_\ell$, denoted $(x_1, \ldots, x_n)$.
Output: An arithmetic sharing modulo 2 of x^{2^m-1} in $\mathbb{Z}_2[X]/\Phi_\ell$, denoted $(y_1, \ldots, y_n)$.

1: $y_1, \ldots, y_n \leftarrow (1, 0, \cdots, 0)$
2: **for** $i = k-1$ to 0 **do**
3: $\quad m' \leftarrow m \gg (i+1)$
4: $\quad$**for** $l = 1$ **to** n **do** $z_l \leftarrow y_l^{2^{m'}}$
5: $\quad z_1, \ldots, z_n \leftarrow \mathsf{Refresh}_{S/2}(z_1, \ldots, z_n)$
6: $\quad y_1, \ldots, y_n \leftarrow \mathsf{SecMult}((y_1, \ldots, y_n), (z_1, \ldots, z_n))$
7: $\quad$**if** $m_i = 1$ **then**
8: $\quad\quad$**for** $l = 1$ **to** n **do** $y_l \leftarrow y_l^2$
9: $\quad\quad y_1, \ldots, y_n \leftarrow \mathsf{SecMult}((y_1, \ldots, y_n), (x_1, \ldots, x_n))$
10: $\quad$**end if**
11: **end for**
12: **return** $y_1, \ldots, y_n$

B.4 Masking inversion in S/q

We provide an algorithmic description of the high-order masked version of the SqInverse algorithm from Section 4.3. As previously, we assume that we have a SecMulPoly algorithm for high-order computing the product of two polynomials in $\mathbb{Z}_q[X]/\Phi_\ell$, with the SNI property, as it can be obtained as a straightforward extension of the And gadget from [ISW03].

Algorithm 21 SecSqInverse$(a_1, \ldots, a_n)$

Input: An arithmetic sharing modulo q $(a_1, \ldots, a_n)$ of $a \in S/q^\times$.
Output: An arithmetic sharing modulo q $(v_1, \cdots, v_n)$ of v such that $v \cdot a = 1 \bmod (q, \Phi_\ell)$.

$v_1, \ldots, v_n \leftarrow \mathsf{SecFastExpo}((a_1 \bmod 2, \ldots, a_n \bmod 2), \ell - 2)$
$v_1, \ldots, v_n \leftarrow (v_1^2 \bmod q, \ldots, v_n^2 \bmod q)$
$t \leftarrow 1$
while $t < \log_2(q)$ **do**
$\quad v'_1, \ldots, v'_n \leftarrow v_1, \ldots, v_n$
$\quad v_1, \ldots, v_n \leftarrow \mathsf{SecMulPoly}((v_1, \ldots, v_n), (-a_1, \ldots, -a_n))$
$\quad v_1 \leftarrow v_1 + 2$
$\quad v_1, \ldots, v_n \leftarrow \mathsf{SecMulPoly}((v'_1, \ldots, v'_n), (v_1, \ldots, v_n))$
$\quad t \leftarrow 2t$
end while
return $(v_1, \ldots, v_n)$

B.5 Proof of Theorem 5

The SecFastExpo algorithm is SNI, thanks to the SNI property of SecMult and the SNI mask refreshing at Line 5. Similarly, the SecSqInverse is SNI, by composition of SNI gadgets.

B.6 Addition chain improvement

The FastExpo algorithm is not the most efficient since it does not necessarily use the minimal addition chain. In particular, for computing an inverse over $\mathbb{Z}_2[X]/\Phi_{701}$ we have the following minimal addition chain for $699: 1 < 2 < 3 < 6 < 12 < 15 < 27 < 42 < 84 <$

$168 < 336 < 672 < 699$. Hence, we deduce the following algorithm computing the inverse with 12 multiplications, instead of 15 multiplications for Algorithm 9, as in [HRSS17].

Algorithm 22 FastInvS2_701(x)

Input: An element $x \in \mathbb{Z}_2[X]/\Phi_{701}$
Output: The inverse of x in $\mathbb{Z}_2[X]/\Phi_{701}$

$y_0 \leftarrow x^2$
$y_1 \leftarrow y_0^2 \times y_0$
$y_2 \leftarrow y_1^2 \times y_0$
$y_3 \leftarrow y_2^{2^3} \times y_2$
$y_4 \leftarrow y_3^{2^6} \times y_3$
$y_5 \leftarrow y_4^{2^3} \times y_2$
$y_6 \leftarrow y_5^{2^{12}} \times y_4$
$y_7 \leftarrow y_6^{2^{15}} \times y_5$
$y_8 \leftarrow y_7^{2^{42}} \times y_7$
$y_9 \leftarrow y_8^{2^{84}} \times y_8$
$y_{10} \leftarrow y_9^{2^{168}} \times y_9$
$y_{11} \leftarrow y_{10}^{2^{336}} \times y_{10}$
$y_{12} \leftarrow y_{11}^{2^{27}} \times y_6$
return y_{12}

We also recall the minimal addition chains for $\ell - 2$ for the four versions of the NTRU parameters (see Table 2):

$$\begin{aligned}
507 &: 1 < 2 < 3 < 6 < 12 < 15 < 30 < 60 < 63 < 126 < 252 < 504 < 507 \\
675 &: 1 < 2 < 3 < 5 < 10 < 20 < 21 < 42 < 84 < 168 < 336 < 672 < 675 \\
699 &: 1 < 2 < 3 < 6 < 12 < 15 < 27 < 42 < 84 < 168 < 336 < 672 < 699 \\
819 &: 1 < 2 < 2 < 6 < 12 < 24 < 48 < 51 < 102 < 204 < 408 < 816 < 819
\end{aligned}$$

We note that the masking of Algorithm 22 is straightforward. It suffices to replace each multiplication with a secure multiplication and apply the linear power-of-two exponentiation on each share independently. However, one should be careful about refreshes when the two shared inputs are linearly dependent.

C Packing $S/3$ polynomials from S/q

During decryption, it is required to pack polynomials with coefficients in $\{0, 1, q-1\}$. In the unmasked version, this is performed by first applying the map $\{0, 1, q-1\} \mapsto \{0, 1, 2\}$ to the five coefficients to obtain $(v_0, \ldots, v_4) \in \{0, 1, 2\}^5$ and then packing as depicted in Section 6.3. While straightforwardly applying the map is cheap in unmasked form, it is more expensive over shares. Instead, we use the following trick: consider the function

$$f : \mathbb{Z}_{512} \to \mathbb{Z}_{512} : x \mapsto x \cdot (511 + 3x)$$

that effectively maps the set $\{0, 1, 511\}$ to $\{0, 2, 4\}$ in $\mathbb{Z}_{512}$. We note that a masked version of f is fairly cheap to compute over arithmetic shares modulo 512 since the only non-linear operation is a SecMult. We first map the coefficients from $\{0, 1, q-1\}$ to $\{0, 1, 511\}$ by reducing every share mod 512 (recall that q is a power of two) and then apply the masked f to bring the coefficients in $\{0, 2, 4\}$ in arithmetic form modulo 512. Once we have our five coefficients $(v'_0, \dots, v'_4) \in \{0, 2, 4\}^5$, we compute

$$x' = \sum_{j=0}^{4} 3^j \cdot v'_j = 2 \cdot \sum_{j=0}^{4} 3^j \cdot v_j$$

as in the regular packS3. Eventually, we obtain the correct result by performing an arithmetic to Boolean conversion of x' and right-shifting every share by 1, effectively dividing x' by 2. We note that it is trivial to find an equivalent to f over $\mathbb{Z}_q$ and thus that we could have directly mapped $\{0, 1, q-1\}$ to $\{0, 2, 4\}$ but we decided to first reduce modulo 512 (which is the smallest power of two giving a result holding over $\mathbb{Z}$) to make the arithmetic to Boolean conversion cheaper.

D High-order computing inverses over $S/3 = \mathbb{Z}[x]/(3, \Phi_\ell)$

D.1 Computing inverses over $S/3$

At Step 5 of KeyGen, we must compute $f_3 = (1/f) \bmod (3, \Phi_\ell)$. Since 3 is of maximal order in $\mathbb{Z}_\ell^\times$, the cyclotomic polynomial Φ_ℓ is irreducible modulo 3 and therefore $S/3$ is a field, with $|S/3| = |\mathbb{Z}_3^{<\ell-1}[X] \setminus \{0\}| = 3^{\ell-1} - 1$. Therefore, as in the modulo 2 case, we can compute the inverse of f via an exponentiation:

$$f^{-1} = f^{3^{\ell-1}-2} = f^{3\cdot(3^{\ell-2}-1)+1} \pmod{(3, \Phi_\ell)}$$

To compute this exponentiation efficiently, we can adapt equation (6) from the modulo 2 case, using the identity $3^{a+b} - 1 = 3^a \cdot (3^b - 1) + (3^a - 1)$:

$$f^{(3^{a+b}-1)} = f^{3^a\cdot(3^b-1)+(3^a-1)} \bmod (3, \Phi_\ell)$$

Adapting Algorithm 9 from Section 4.3, we obtain the following algorithm. The correctness is proved similarly.

Algorithm 23 FastExpo3(x, m)

Input: An integer $m = (m_{k-1}, \dots, m_0)_2$ and an element $x \in \mathbb{Z}_3[X]/\Phi_\ell$
Output: $x^{(3^m-1)}$ in $\mathbb{Z}_3[X]/\Phi_\ell$

1: $y \leftarrow 1$
2: $x \leftarrow x \times x$
3: **for** $i = k-1$ to 0 **do**
4: $\quad m' \leftarrow m \gg (i+1)$
5: $\quad y \leftarrow y \times y^{3^{m'}}$
6: $\quad$**if** $m_i = 1$ **then** $y \leftarrow y^3 \times x$
7: **end for**
8: **return** y

D.2 High-order inversion in $S/3$

We describe the high-order masking of the previous FastExpo3 algorithm.

Algorithm 24 $\mathsf{SecFastExpo3}(x, m)$

Input: An integer $m = (m_{k-1}, \ldots, m_0)_2$ and an arithmetic sharing modulo 3 $(x_1, \ldots, x_n)$ of an element $x \in \mathbb{Z}_3[X]/\Phi_\ell$

Output: An arithmetic sharing modulo 3 $(y_1, \ldots, y_n)$ of $x^{(3^m-1)}$ in $\mathbb{Z}_3[X]/\Phi_\ell$

$y_1, \ldots, y_n \leftarrow (1, 0, \ldots, 0)$
$x'_1, \ldots, x'_n \leftarrow \mathsf{Refresh}_{\mathbb{Z}_3}(x_1, \ldots, x_n)$
$x_1, \ldots, x_n \leftarrow \mathsf{SecMult}((x_1, \ldots, x_n), (x'_1, \ldots, x'_n))$
for $i = k-1$ to 0 **do**
    $m' \leftarrow m \gg (i+1)$
    for $l = 1$ **to** n **do** $z_l \leftarrow y_l^{3^{m'}}$
    $z_1, \ldots, z_n \leftarrow \mathsf{Refresh}_3(z_1, \ldots, z_n)$
    $y_1, \ldots, y_n \leftarrow \mathsf{SecMult}((y_1, \ldots, y_n), (z_1, \ldots, z_n))$
    if $m_i = 1$ **then**
        for $l = 1$ **to** n **do** $y_l \leftarrow y_l^3$
        $y_1, \ldots, y_n \leftarrow \mathsf{SecMult}((y_1, \ldots, y_n), (x_1, \ldots, x_n))$
    end if
end for
return $y_1, \ldots, y_n$

The theorem below shows our inverse algorithm $\mathsf{SecFastExpo3}$ achieves the $t-\mathsf{SNI}$ security notion. The proof is similar to the proof of Theorem 5 and is therefore omitted.

Theorem 9 ($t-\mathsf{SNI}$ security of $\mathsf{SecFastExpo3}$)**.** *For any subset $O \subset [1, n]$ and any t_1 intermediate variables with $t_1 + |O| \leq t$, the output variables $y_{|O}$ and the t_1 intermediate variables can be perfectly simulated from input variables $x_{|I}$, with $|I| \leq t_1$.*