Improved High-Order Masked Generation of Masking Vector and Rejection Sampling in Dilithium

Jean-Sébastien Coron[1], François Gérard[1], Tancrède Lepoint[2], Matthias Trannoy[1,3], and Rina Zeitoun[3]

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

Abstract In this work, we introduce enhanced high-order masking techniques tailored for Dilithium, the post-quantum signature scheme recently standardized by NIST. We improve the masked generation of the masking vector $\mathbf{y}$, based on a fast Boolean-to-arithmetic conversion modulo q. We also describe an optimized gadget for the high-order masked rejection sampling, with a complexity independent from the size of the modulus q. We prove the security of our gadgets in the classical ISW t-probing model. Finally, we detail our open-source C implementation of these gadgets integrated into a fully masked Dilithium implementation, and provide an efficiency comparison with previous works.

1 Introduction

Dilithium signatures. The Dilithium signature scheme [BDK+21] was recently announced in 2022 as the primary signature scheme standardized by the National Institute of Standards and Technologies (NIST) in their post-quantum algorithm competition. Dilithium is a lattice-based scheme utilizing the "Fiat-Shamir with Aborts" technique developed by Lyubashevsky [Lyu09] based on rejection sampling, so that the signature does not leak information about the secret-key. We recall in Fig. 1 the mechanism of Dilithium signatures.

In recent years, many side-channels attacks against Dilithium have been described, see for example [BDE+18,LZS+21,MUTS22], underscoring the need for robust side-channel countermeasures. Below we recall the modern approach for protecting a cryptographic algorithm against side-channel attacks, namely the masking countermeasure and the ISW probing model with the NI/SNI definitions.

The masking countermeasure. The goal of the masking countermeasure is to split every secret-dependent variable x into n shares x_i with $x = x_1 \oplus \cdots \oplus x_n$, so that an adversary observing a fraction of the computation does not get information about the secret-key. To put the masking countermeasure into firm scientific grounds, [ISW03] introduced the t-probing model, in which an attacker can probe up to t variables. The authors showed that using $n = 2t + 1$ shares, any Boolean circuit C of size $|C|$ can be transformed into an equivalent circuit $\tilde{C}$ that is t-probing secure, with a complexity quadratic in n. Throughout this paper, we will use the standard NI/SNI definitions introduced in [BBD+15] that facilitate the writing of security proofs. Namely, one can focus on proving the NI/SNI security of individual masking gadgets, and the security of the full algorithm follows by composition.

Since lattice-based cryptography combines Boolean and arithmetic operations, it is usually more efficient to mask some variables with Boolean masking $x = x_1 \oplus \cdots \oplus x_n$, and some other variables with arithmetic masking $x = x_1 + \cdots + x_n \bmod q$, rather than applying ISW generically with Boolean masking only. This approach requires frequent conversions between the two representations. The first high-order conversion between arithmetic and Boolean masking was described in [CGV14] for a power-of-two modulus q. It was then extended to any modulus q in [BBE+18]. In [SPOG19], the authors described an efficient 1-bit Boolean to arithmetic conversion modulo q, which is easily converted into a k-bit Boolean to arithmetic conversion. In [BCZ18], the authors described a k-bit Boolean to arithmetic modulo 2^k conversion, with complexity $\mathcal{O}(2^n)$, but independent from k when using k-bit registers; in practice, for small values of n, the algorithm is significantly faster than alternative algorithms.

```
KeyGen
1: $\mathbf{A} \leftarrow R_q^{k\times\ell}$
2: $(\mathbf{s}_1, \mathbf{s}_2) \leftarrow S_\eta^\ell \times S_\eta^k$
3: $\mathbf{t} := \mathbf{A}\mathbf{s}_1 + \mathbf{s}_2$
4: return $(pk = (\mathbf{A}, \mathbf{t}_1),\ sk = (\mathbf{A}, \mathbf{t}, \mathbf{s}_1, \mathbf{s}_2))$

Sign$(sk, M)$
1: $\mathbf{z} := \perp$
2: while $\mathbf{z} = \perp$ do
3:     $\mathbf{y} \leftarrow \tilde{S}_{\gamma_1}^\ell$
4:     $\mathbf{w} := \mathbf{A}\mathbf{y}$
5:     $(\mathbf{w}_0, \mathbf{w}_1) := \mathsf{Decompose}_q(\mathbf{w}, 2\gamma_2)$
6:     $c := H(M \,\|\, \mathbf{w}_1)$
7:     $\mathbf{z} := \mathbf{y} + c\mathbf{s}_1$
8:     $\tilde{\mathbf{r}} := \mathbf{w}_0 - c\mathbf{s}_2$
9:     if $\|\mathbf{z}\|_\infty \geq \gamma_1 - \beta$ or $\|\tilde{\mathbf{r}}\|_\infty \geq \gamma_2 - \beta$, then $\mathbf{z} := \perp$
10: end while
11: return $\sigma = (\mathbf{z}, c)$

Verify$(pk, M, \sigma = (\mathbf{z}, c))$
1: $\mathbf{w}_1' := \mathsf{HighBits}(\mathbf{A}\mathbf{z} - c\mathbf{t})$
2: return $[\![\|\mathbf{z}\|_\infty < \gamma_1 - \beta]\!]$ and $[\![c = H(M \,\|\, \mathbf{w}_1')]\!]$
```

Figure 1. Simplified template of Dilithium with $\tilde{\mathbf{r}}$-version (reference implementation), without public-key compression.

Masking Dilithium signatures. The high-order masking of lattice-based signatures was initiated in [BBE+18] with the masking of the GLP scheme [GLP12], a predecessor of Dilithium. Subsequent efforts to implement high-order masking for Dilithium were initially detailed in [MGTF19] and later refined in [ABC+23]. In particular, the vector $\mathbf{w} = \mathbf{A}\mathbf{y}$ (Line 4) should be masked, while it was left unmasked in [MGTF19], which could potentially lead to the recovery of the private key. The authors of [ABC+23] also argued that $\mathbf{w}_1$ in $(\mathbf{w}_1, \mathbf{w}_0) \leftarrow \mathsf{Decompose}_q(\mathbf{w})$ (Line 5) can be computed in the clear because it is also publicly computed during signature verification. Conveniently, this implies that the challenge $\tilde{c} = H(M\|\mathbf{w}_1)$ need not be masked, thereby eliminating the need to mask the Keccak hash function H. Similarly, for Dilithium signature with public-key compression, the variable $\mathbf{w} - c\mathbf{s}_2 = \mathbf{A}\mathbf{z} - c\mathbf{t}$ is public after the rejection sampling. Therefore, as explained in [ABC+23], one does not need to mask the computation of the hint vector $\mathbf{h}$. We follow the same approach in this paper.

Therefore, starting from the generation of $\mathbf{y}$ at Line 3 in Fig. 1, the variables $\mathbf{y}$, $\mathbf{w}$, $\mathbf{w}_0$, $\mathbf{z}$ and $\tilde{\mathbf{r}}$ must all be masked. This implies that the following operations must be masked:

- Masked generation of $\mathbf{y} \leftarrow \tilde{S}_{\gamma_1}^{\ell}$ (Line 3). Each coefficient y of the masking vector $\mathbf{y}$ must be uniformly distributed in an interval. Following [BBE+18], this operation can be masked by first generating a random μ-bit Boolean masked coefficient, and then performing a Boolean to arithmetic conversion modulo q, using for example [SPOG19].
- Masking of Decompose (Line 5). As explained above, only $\mathbf{w}_0$ in $(\mathbf{w}_1, \mathbf{w}_0) \leftarrow \mathsf{Decompose}_q(\mathbf{w})$ must be masked. In [ABC+23], the authors described an efficient masking of Decompose, which was later improved in [CGTZ23].
- Masked rejection sampling (Line 9). The variable $\mathbf{z}$ can only be unmasked after successfully completing the rejection sampling of both $\mathbf{z}$ and $\tilde{\mathbf{r}}$ to prevent leakage of the secret-key. The approach in [BBE+18] uses integer comparison, leveraging arithmetic to Boolean masking for sign bit extraction. The same approach is used in [ABC+23] with an improved algorithm, with fewer operations than in [BBE+18].

Recently, for the masked generation of the masking vector $\mathbf{y}$, the authors of [CGTZ23] have described an improved Boolean to arithmetic conversion, with complexity independent from the Boolean input size and modulus size. Their technique is based on first performing a fast Boolean to arithmetic conversion modulo 2^k using [BCZ18] for a certain parameter k, and then performing a modulus switching to modulo q; to correct the error introduced in the modulus switching, one eventually performs a sequence of arithmetic shifts. Their algorithm therefore extends the fast Boolean to arithmetic conversion of [BCZ18], previously limited to power-of-two moduli (see Table 1).

	Direction	Complexity	Register size
[CGV14]	B $\rightarrow$ A (mod 2^k)	$\mathcal{O}(n^2 \cdot k)$	–
[BBE+18]	B $\rightarrow$ A (mod q)	$\mathcal{O}(n^2 \cdot \log q)$	–
[SPOG19]	B $\rightarrow$ A (mod q)	$\mathcal{O}(n^2 \cdot \mu)$	–
[BCZ18]	B $\rightarrow$ A (mod 2^k)	$\mathcal{O}(2^n)$	–
[CGTZ23]	B $\rightarrow$ A (mod q)	$\mathcal{O}(2^n)$	$k = \mu + \lceil \log_2 q \rceil + \lceil \log_2 n \rceil$
This paper	**B $\rightarrow$ A (mod q)**	$\mathcal{O}(2^n)$	$k = \mu + \lceil \log_2 n \rceil$

Table 1. Complexities of μ-bit Boolean to arithmetic conversions with n shares.

Our contributions. This paper introduces enhanced high-order masking techniques tailored for Dilithium, to improve runtime complexity. We make the following contributions:

- We enhance the fast Boolean to arithmetic conversion algorithm of [CGTZ23] for high-order masking the generation of the masking vector $\mathbf{y}$ in Dilithium. While maintaining the same asymptotic complexity $\mathcal{O}(2^n)$, our improvement reduces the register size to $k = \mu + \lceil \log_2 n \rceil$ from the original $k = \mu + \lceil \log_2 q \rceil + \lceil \log_2 n \rceil$, where μ represents the Boolean input's bit size; see Table 1. This adjustment allows us to use 32-bit registers instead of 64-bit ones for Dilithium's parameters. Consequently, on a 32-bit processor, our optimized conversion is expected to be about twice as fast.

- We introduce a new technique for performing the high-order rejection sampling with complexity independent from the size of the modulus q. The new complexity only depends on the size β of the interval that must be rejected by the rejection sampling. Our technique is based on first performing an arithmetic shift of the polynomial coefficient, and then a zero test, for which we use the fast zero-testing procedure from [CGMZ23], whose complexity is independent from the size of q.

Finally, we present in Section 5 a complete high-order masked implementation of Dilithium, with the improved gadgets described above. We provide the practical results of an open source C implementation and compare the performance improvement provided by our new gadgets with those from [ABC+23] and [CGTZ23]. We show that our techniques achieve a notable speedup compared to previous work. Specifically, our new Boolean-to-arithmetic masking algorithm for generating the masked vector $\mathbf{y}$ is faster and easier to implement than that of [CGTZ23], and our enhanced gadget for rejection sampling surpasses the speed of the gadget presented in [ABC+23] for a non-bitsliced implementation, which itself represented a significant advancement over the original method in [BBE+18]. The plain C code is publicly available at

`https://github.com/fragerar/tches24_masked_Dilithium`

Dilithium vs ML-DSA. The main difference between Dilithium (Version 3.1) [BDK+21] and the ML-DSA draft standard [NIS23] centers on the seed ρ' generation for the vector $\mathbf{y}$. Dilithium uses a 512-bit random string for ρ' in its randomized signatures, while ML-DSA adopts a "hedged" approach, generating ρ' pseudo-randomly using the signer's private-key, the message M, and a 256-bit random string rnd, thereby maintaining security even if the RNG is compromised.

In randomized Dilithium, the coefficients y in $\mathbf{y}$ are computationally indistinguishable from the uniform distribution within $]-\gamma_1, \gamma_1]$, so we can directly generate each masked coefficient y uniformly, as in [ABC+23,CGTZ23], bypassing the need to mask the hash function H and the ExpandMask function. Instead, for our fully masked ML-DSA implementation detailed in Section 5.4, we incorporate masked H and ExpandMask functions to maintain ML-DSA's "hedged" feature. We then compare its performances with randomized Dilithium signatures.

2 Notations and security definitions

2.1 Notations

We adopt the same notations as the Dilithium specifications [BDK+21]. Let $\mathbb{Z}_q$ denote the ring of integers modulo q. We define the polynomial quotient rings $R = \mathbb{Z}[X]/(X^{256}+1)$ and $R_q = \mathbb{Z}_q[X]/(X^{256}+1)$. The infinity norm of an element $z = \sum z^{(i)} X^i \in R_q$ is denoted by $\|z\|_\infty$ and defined as follows:

$$\|z\|_\infty = \max_{0 \le i < 256} |z^{(i)} \bmod^{\pm} q|$$

where for $x \in \mathbb{Z}$, we denote by $x \bmod^{+} q$ (resp. $x \bmod^{\pm} q$) the positive (resp. centered) representative of x modulo q. We denote by $\tilde{S}_\eta$ the polynomials of R with coefficients in the range $]-\eta, \eta]$.

We use bold lower-case letters to represent column vectors with coefficients in R or R_q. For a vector of polynomials $\mathbf{z} = (z_1, \ldots, z_k) \in R_q^k$, the infinity norm is defined as $\|\mathbf{z}\|_\infty = \max_{1 \le i \le k} \|z_i\|_\infty$.

2.2 Security definitions

In the following, we review the Non-Interference (NI) and Strong Non-Interference (SNI) security notions introduced in [BBD+16]. The main advantage of these notions is that the security of individual gadgets can be proven NI or SNI, with the probing security of the full circuit subsequently being deduced through composition.

Definition 1 (t-NI security). *Let G be a gadget taking as input n shares $(a_1, \ldots, a_n)$ and outputting n shares $(b_1, \ldots, b_n)$. The gadget G is said to be t-NI secure if for any set of $t_1 \le t$ probed variables there exists a subset of input indices $I \subset [1, n]$ such that the t_1 probed variables can be perfectly simulated from $a_{|I}$, with $|I| \le t_1$.*

Definition 2 (t-SNI security). *Let G be a gadget taking as input n shares $(a_1, \ldots, a_n)$ and outputting n shares $(b_1, \ldots, b_n)$. The gadget G is said to be t-SNI secure if for any set of t_1 intermediate variables and any subset $O \subset [1, n]$ of output indices such that $t_1 + |O| \le t$, there exists a subset of input indices $I \subset [1, n]$ such that the t_1 intermediate variables and the outputs $b_{|O}$ can be perfectly simulated from $a_{|I}$, with $|I| \le t_1$.*

For masking Dilithium, the masked output of the rejection sampling must eventually be recombined and computed in the clear. For this we use the extended notion of NI security from [BBE+18, Definition 7], in which the output b of the gadget is given to the simulator.

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

3 Improved masked generation of the vector y in Dilithium

3.1 Existing work

In Dilithium, the generation of the masking vector $\mathbf{y} \leftarrow \tilde{S}_{\gamma_1}^{\ell}$ must be masked (see Line 3 in Fig. 1). The goal is to obtain arithmetic shares $\mathbf{y}_1, \ldots, \mathbf{y}_n$ of $\mathbf{y}$:

$$\mathbf{y} = \mathbf{y}_1 + \ldots + \mathbf{y}_n \pmod{q}$$

so that the next operations $\mathbf{w} := \mathbf{A}\mathbf{y}$ (Line 4) and $\mathbf{z} := \mathbf{y} + c\mathbf{s}_1$ (Line 7) are easily masked arithmetically modulo q. Note that we cannot simply generate $\mathbf{y}$ and then the arithmetic shares $\mathbf{y}_i$ satisfying the above equation, as the adversary could directly probe $\mathbf{y}$. Instead, we must generate the arithmetic shares $\mathbf{y}_i$ without leaking information about $\mathbf{y}$.

Deterministic vs randomized Dilithium. For Dilithium signatures, Fig. 1 only provides a simplified template. In particular, in the Dilithium specification [BDK+21], the masking vector $\mathbf{y}$ is generated by expanding a seed ρ' with $\mathbf{y} \in \tilde{S}^{\ell}_{\gamma_1} := \mathsf{ExpandMask}(\rho', \kappa)$. In the *deterministic* version of Dilithium, the seed ρ' is generated pseudo-randomly from the user's private-key and the message M, with $\rho' = H(K \| H(tr \| M))$. In the *randomized* version of Dilithium, the seed ρ' is instead sampled as a 512-bit random string. In both cases, the function ExpandMask is used to deterministically generate the randomness of each coefficient y of $\mathbf{y} \in \tilde{S}^{\ell}_{\gamma_1}$, from the seed ρ'.

For the randomized version of Dilithium, the distribution of each coefficient y of $\mathbf{y}$ is therefore computationally indistinguishable from uniform in the interval $]-\gamma_1, \gamma_1]$. In that case, for the masked version, we follow the same approach as in [ABC+23,CGTZ23]: we generate each masked coefficient y directly with the uniform distribution in the interval $]-\gamma_1, \gamma_1]$, without using the ExpandMask function. However, for the deterministic version of Dilithium, we must mask the generation of $\rho' = H(K \| H(tr \| M))$. For this, we start from the shares $K_1, \ldots, K_n$ of the private-key element K, and we obtain the Boolean shares ρ'_i of the seed ρ'; the function ExpandMask must also be masked.

Masked generation of y. We recall the approach initiated in [BBE+18] for generating a masked $\mathbf{y}$, based on Boolean to arithmetic masking. By definition, each coefficient y of the masking vector $\mathbf{y}$ must be generated in the interval $]-\gamma_1, \gamma_1]$, with $\gamma_1 = 2^{17}$ for Security Level 2. This interval has length 2^{μ} for $\mu = \log_2 \gamma_1 + 1 = 18$. Therefore, for each coefficient of $\mathbf{y}$, we first compute the shares u_i of a μ-bit Boolean masked x:

$$x = u_1 \oplus \cdots \oplus u_n$$

In the deterministic version of Dilithium, these shares u_i are obtained from the masked Expand-Mask as explained above. For the randomized Dilithium, we generate each μ-bit Boolean share u_i independently at random.

One can then proceed with a Boolean to arithmetic conversion algorithm, which gives arithmetic shares y_i such that $x = y_1 + \cdots + y_n \pmod{q}$. Eventually, to get the uniform distribution in $]-\gamma_1, \gamma_1]$ instead of $[0, 2\gamma_1[$, it suffices to offset the first share y_1 by $\gamma_1 - 1$. The Boolean to arithmetic conversion algorithm from [SPOG19] has complexity $\mathcal{O}(\mu \cdot n^2)$, therefore the full procedure has complexity $\mathcal{O}(\mu \cdot n^2)$.

Fast Boolean to arithmetic conversion modulo *q*. In [CGTZ23], the authors introduced a fast high-order Boolean to arithmetic conversion modulo q, with complexity independent from μ and the size of q. The technique is based on modulus switching: one first performs a fast Boolean to arithmetic conversion modulo 2^k using [BCZ18] for a certain parameter k, and then a modulus switching modulo q; eventually, to get rid of the error introduced by the modulus switching, one eventually performs a sequence of arithmetic shifts.

As previously, one starts with random μ-bit shares u_i, which encode a random μ-bit integer $x = u_1 \oplus \cdots \oplus u_n$. In the second step, one applies the conversion of [BCZ18] from a μ-bit Boolean masking into an arithmetic modulo 2^k masking, for a certain parameter k, which gives:

$$x = x_1 + \cdots + x_n \pmod{2^k}$$

The advantage of the conversion from [BCZ18] is that its complexity is independent from the modulus size k, and although its asymptotic complexity is $\mathcal{O}(2^n)$, it is quite efficient for small values of n, at least one order of magnitude more efficient than [SPOG19]. As shown in [CGTZ23], by performing a modulus switching, one can then obtain arithmetic shares y_i

$$y_1 + \cdots + y_n = 2^\alpha \cdot x + e \pmod{q 2^\alpha}$$

for a small error $0 \leq e < n$, where we take $\alpha = \lceil \log_2 n \rceil$, so that $0 \leq e < 2^\alpha$. Eventually, the error e introduced by the modulus switching is removed by performing an arithmetic shift by α bits, so that we obtain the arithmetic shares

$$x = z_1 + \cdots + z_n \pmod{q}$$

as required. As previously, to get the uniform distribution in $]-\gamma_1, \gamma_1]$, it suffices to offset the first share z_1 by $\gamma_1 - 1$. The total complexity remains $\mathcal{O}(2^n)$ and is independent of the modulus size, as in [BCZ18]. The Boolean to arithmetic conversion algorithm from [CGTZ23] is therefore an extension of [BCZ18] to arbitrary moduli q.

However, the above conversion algorithm requires a relatively large register size k. As shown in [CGTZ23], one must use $k = \mu + \lceil \log_2 q \rceil + \lceil \log_2 n \rceil$ in the initial Boolean to arithmetic conversion modulo 2^k from [BCZ18]. For Dilithium with Security Level 2, with $q = 2^{23} - 2^{13} + 1$ and $\mu = 18$, this gives $k \geq 41$. This implies that within the [BCZ18] conversion algorithm, we must work with 64-bit registers.

In the next section, we describe a variant of the above Boolean to arithmetic conversion algorithm with roughly the same number of operations, but with a smaller register size $k = \mu + \lceil \log_2 n \rceil$. Therefore, the new register size only depends on μ and not on the size of q; see Table 1 for a summary of Boolean to arithmetic conversions. For the Dilithium parameters, this enables to work with 32-bit registers instead of 64-bit registers.

3.2 Our new high-order Boolean to arithmetic algorithm modulo q

In the following, we describe our alternative algorithm for performing the high-order Boolean to arithmetic algorithm modulo q, with the same complexity as the algorithm from [CGTZ23] recalled above, but with a smaller register size k. As previously, we are given as input a μ-bit Boolean masking $x = u_1 \oplus \cdots \oplus u_n$, and our goal is to obtain an arithmetic masking modulo q of x. We first perform a Boolean to arithmetic conversion using [BCZ18], which gives:

$$x = x_1 + \cdots + x_n \pmod{2^k}$$

for a parameter $k := \mu + \alpha$ with $\alpha := \lceil \log_2 n \rceil$. From the previous equation, we can write $x_1 + \cdots + x_n = x + \delta \cdot 2^k$ for some $\delta \in \mathbb{Z}$. This later equation holds over $\mathbb{Z}$, therefore it also holds modulo $2^k q$, which provides an arithmetic sharing of $z := x + \delta \cdot 2^k$:

$$z = x_1 + \cdots + x_n = x + \delta \cdot 2^k \pmod{2^k q} \tag{1}$$

Our goal is now to derive an arithmetic sharing of δ modulo q, from which it is easy to derive an arithmetic sharing of x modulo q.

We proceed as follows. We first perform a modulus switching to get an arithmetic sharing of $2^\alpha \delta$ modulo $2^\alpha q$, up to some additive error. For this, we apply the following modulus switching lemma [CGTZ23].

Lemma 1 (Modulus switching [CGTZ23]). *Let p_1 and p_2 be two positive integers. Let $x_i \in \mathbb{Z}_{p_1}$ for $1 \le i \le n$ and let $x = x_1 + \cdots + x_n \pmod{p_1}$. Let $y_1 := \lfloor x_1 \cdot p_2/p_1 \rfloor + n - 1 \bmod p_2$ and $y_i := \lfloor x_i \cdot p_2/p_1 \rfloor \bmod p_2$ for $2 \le i \le n$. Then, for some $0 \le e \le n-1$:*

$$y_1 + \cdots + y_n = \left\lfloor \frac{x \cdot p_2}{p_1} \right\rfloor + e \pmod{p_2} \tag{2}$$

From the shares x_i of z in (1), by applying the above Lemma with $p_1 := 2^k q$ and $p_2 := 2^\alpha q$, from $p_2/p_1 = 2^{\alpha-k} = 1/2^\mu$, we obtain an arithmetic sharing of $\lfloor z/2^\mu \rfloor$ up to some additive error $0 \le e < n$:

$$y_1 + \cdots + y_n = \left\lfloor \frac{z}{2^\mu} \right\rfloor + e \pmod{2^\alpha q}$$

Moreover, from $z = x + \delta \cdot 2^k$ and $0 \le x < 2^\mu$, we have $\lfloor z/2^\mu \rfloor = \delta \cdot 2^{k-\mu} = \delta \cdot 2^\alpha$. Therefore, from the previous equation we obtain an arithmetic sharing of $\delta \cdot 2^\alpha$, up to a small additive error e:

$$y_1 + \cdots + y_n = \delta \cdot 2^\alpha + e \pmod{2^\alpha q} \tag{3}$$

Recall that the error e satisfies $0 \le e < n \le 2^\alpha$. Therefore, by performing an arithmetic shift by α bits, we can get rid of the error e and obtain an arithmetic sharing of $\delta + \lfloor e/2^\alpha \rfloor = \delta$ modulo q. For this, we apply the $\mathsf{ShiftMod}_\beta$ algorithm from [CGTZ23], which we recall in Appendix A.

Theorem 1 ([CGTZ23]). *The $\mathsf{ShiftMod}_\beta$ algorithm, taking as input an n-arithmetic sharing of x modulo $q \cdot 2^\beta$, outputs an n-arithmetic sharing of $\lfloor x/2^\beta \rfloor$ modulo q, in time $\mathcal{O}(\beta \cdot n^2)$.*

Therefore, by applying the $\mathsf{ShiftMod}_\beta$ algorithm with $\beta = \alpha$ on the shares y_i from (3), we obtain as required an arithmetic sharing with shares δ_i satisfying:

$$\delta_1 + \cdots + \delta_n = \left\lfloor \frac{\delta \cdot 2^\alpha + e}{2^\alpha} \right\rfloor = \delta \pmod{q}$$

Eventually, from (1) we obtain:

$$x = \sum_{i=1}^{n} x_i - \delta \cdot 2^k = \sum_{i=1}^{n} x_i - 2^k \cdot \sum_{i=1}^{n} \delta_i = \sum_{i=1}^{n} \left(x_i - 2^k \cdot \delta_i \right) \pmod{q}$$

which gives an arithmetic masking of x modulo q, as required. We provide in Alg. 1 below the formal description of the corresponding algorithm. We use a SNI mask refreshing algorithm $\mathsf{RefreshMasks}$, see for example [BBD+16].

Algorithm 1 Boolean to Arithmetic conversion ($\mathsf{BtoA_qDelta}$)

Input: A modulus q, a μ-bit Boolean masking $u_1, \ldots, u_n$ of x such that $u_1 \oplus \cdots \oplus u_n = x$
Output: An arithmetic sharing $v_1, \ldots, v_n$ such that $v_1 + \cdots + v_n = x \pmod{q}$

```
1: $\alpha \leftarrow \lceil \log_2 n \rceil$
2: $k \leftarrow \mu + \alpha$
3: $x_1, \ldots, x_n \leftarrow \mathsf{BtoAExp}_{\mu, 2^k}(u_1, \ldots, u_n)$
4: for $i = 1$ to $n$ do $y_i \leftarrow \lfloor x_i / 2^\mu \rfloor$
5: $y_1 \leftarrow y_1 + n - 1$
6: $(\delta_1, \ldots, \delta_n) \leftarrow \mathsf{ShiftMod}_\alpha(2^\alpha \cdot q, (y_1, \ldots, y_n))$
7: $(\delta_1, \ldots, \delta_n) \leftarrow \mathsf{RefreshMasks}(\delta_1, \ldots, \delta_n)$
8: for $i = 1$ to $n$ do $v_i \leftarrow x_i - 2^k \cdot \delta_i \bmod q$
9: return $(v_1, \ldots, v_n)$
```

Complexity. The number of operations of $\mathsf{BtoAExp}_{\mu,2^k}$ using [BCZ18] is $T_{\mathsf{BtoAExp}} = 10 \cdot 2^n - 6n - 13$. Furthermore, the For loops at lines 4 and 8 cost respectively n and $2n$ operations. The cost of $\mathsf{ShiftMod}_\alpha$ at Line 6 is $\alpha \cdot (2n^2 + 10n - 9)$ operations (see Appendix A). Eventually, the cost of RefreshMasks is $3n(n-1)/2$. Therefore the overall complexity is $T_{\mathsf{BtoA_qDelta}} = 10 \cdot 2^n - 6n - 13 + n + 1 + \alpha \cdot (2n^2 + 10n - 9) + 3n(n-1)/2 + 2n = 10 \cdot 2^n + 3n^2 - 9n/2 - 12 + \alpha \cdot (2n^2 + 10n - 9)$ and asymptotically the number of operation is $\mathcal{O}(2^n + \alpha \cdot n^2) = \mathcal{O}(2^n + n^2 \log n)$ which is still $\mathcal{O}(2^n)$ in total. The complexity of $\mathsf{BtoA_qDelta}$ is therefore very similar to the complexity of $\mathsf{BtoA_qExact}$ from [CGTZ23], but with a smaller value of k within the [BCZ18] conversion, that is $k = \mu + \lceil \log_2 n \rceil$ instead of $k = \mu + \lceil \log_2 q \rceil + \lceil \log_2 n \rceil$. Recall that 2^μ is the length of the interval $]-\gamma_1, \gamma_1]$ in which each coefficient y must be generated, with $\mu = \log_2 \gamma_1 + 1 = 18$ for Security Level 2. With $q = 2^{23} - 2^{13} + 1$ and $\mu = 18$, we get $k = 18 + \lceil \log_2 n \rceil$, and we can therefore use 32-bit registers instead of 64-bit registers.

Theorem 2. *The $\mathsf{BtoA_qDelta}$ algorithm, given a modulus q and a μ-bit Boolean masking $u_1, \ldots, u_n$ of x, outputs an arithmetic sharing $v_1, \ldots, v_n$ of x modulo q in time $\mathcal{O}(2^n)$.*

Security. The following theorem shows that our new Boolean to arithmetic conversion algorithm achieves the NI property.

Theorem 3 **($(n-1) - \mathsf{NI}$ of $\mathsf{BtoA_qDelta}$).** *For any set of t_1 probed variables, there exists a subset $I \subset [1, n]$, with $|I| \leq t_1$, of input indexes such that the t_1 probed variables can be perfectly simulated from $u_{|I}$*

Proof. The NI property follows from the composition of the NI gadget ShiftMod with the SNI gadgets RefreshMasks and BtoAExp.

3.3 Comparison

We provide a comparison of the operation count between the various methods. We assume that we are working with a 32-bit processor, so for the internal conversion method [BCZ18] employed by [CGTZ23], we adjust its complexity from $T_{\mathsf{BtoAExp}}(n) = 10 \cdot 2^n - 6n - 13$ to $2 \cdot T_{\mathsf{BtoAExp}}(n)$, reflecting the necessity to accommodate 64-bit integers. We see that our new algorithm outperforms [CGTZ23] for all orders, and is faster than [SPOG19] and [BBE+18] for small orders. We also provide in Section 5.1 a concrete comparison on Cortex-M4 microcontroller.

B → A mod q	Security order t							
	2	3	4	5	6	8	10	12
[BBE+18] 18 → mod q	2 841	5 215	8 782	12 897	17 825	30 235	46 012	64 776
[SPOG19] 18 → mod q	804	1 414	2 186	3 120	4 216	6 894	10 220	14 194
[CGTZ23] 18 → mod q	194	396	857	1 587	2 969	11 132	42 240	165 572
Algorithm 1	146	280	596	1 024	1 787	6 161	21 972	83 939

Table 2. Operation count for 18-bit Boolean to arithmetic modulo q conversion algorithms, up to security order $t = 12$, with $n = t + 1$ shares, for prime $q = 2^{23} - 2^{13} + 1$.

4 Improved masked rejection sampling for Dilithium

4.1 Existing work

In Dilithium, the rejection sampling consists in restarting the signature generation if $\|\mathbf{z}\|_\infty \geq \gamma_1 - \beta$ or $\|\tilde{\mathbf{r}}\|_\infty \geq \gamma_2 - \beta$ (see Line 9 in Fig. 1); such operation is necessary to prevent any leakage of the secret-key. For a masked Dilithium implementation, this check must be performed over masked values of $\mathbf{z}$ and $\tilde{\mathbf{r}}$. Namely, the shares of $\mathbf{z}$ can only be recombined after the complete rejection sampling of $\mathbf{z}$ and $\tilde{\mathbf{r}}$; otherwise this may leak some information about the secret-key.

We first recall how such rejection sampling is performed in [BBE+18]. For any coefficient x of $\mathbf{z}$ and $\tilde{\mathbf{r}}$, the rejection sampling consists in ensuring that $-a < x < a$, for a known bound a. In [BBE+18], for high-order computing the comparison $x < a$, the authors first convert the arithmetic sharing of x modulo q to Boolean, with a representation $0 \leq x < q$. However, one needs a centered representation $-(q-1)/2 < x \leq (q-1)/2$. This is obtained by high-order computing $x - (q-1)/2$ and $x - q$ on Boolean values, and selecting either x or $x - q$ depending on the most significant bit of $x - (q-1)/2$, which is accomplished by two additional high-order secure And's. Eventually, one computes the subtraction $x - a$ over Boolean values, and output the (high-order masked) most significant bit, which is equal to 1 iff $x < a$. To ensure $|x| < a$, the same procedure is applied to $-x$. The complexity is therefore $T_{\mathsf{RS}} = 2\left(T_{\mathsf{ABmodq}} + 3 \cdot T_{\mathsf{SecAdd}} + 3 \cdot T_{\mathsf{SecAnd}} + 2 \cdot T_{\mathsf{Refresh}} + 3n\right) + T_{\mathsf{SecAnd}}$.

The authors of [ABC+23] described a more efficient method. To test that $|x \bmod^\pm q| < a$, they first compute an arithmetic sharing of $x' = x + a - 1 \bmod q$, and so one must test whether $0 \leq x' < 2a - 1$. For this, they perform an arithmetic to $(k+1)$-bit Boolean conversion of x', with a parameter $k = \lceil \log_2 q \rceil$. From this Boolean representation, they compute using addition over Boolean shares a Boolean sharing of $x'' = x' - 2a \bmod^\pm 2^{k+1}$. One can show that $|x \bmod^\pm q| < a \Leftrightarrow x'' < 0$. To test whether x'' is negative, it suffices to unmask the $(k+1)$-th bit of x''. The complexity is therefore $T_{\mathsf{RS}} = T_{\mathsf{ABmodq}} + T_{\mathsf{SecAdd}}$. We summarize in Table 3 below the corresponding complexities.

	[BBE+18]	[ABC+23]	**This paper**
Complexity	$\mathcal{O}(n^2 \cdot \log q)$	$\mathcal{O}(n^2 \cdot \log q)$	$\mathcal{O}(n^2 \cdot \log \beta n)$

Table 3. Complexities of rejection sampling for a single polynomial coefficient, as a function of the number of shares n, where q is the modulus, and β is the size of the interval that must be rejected.

4.2 Improved masked rejection sampling based on zero-testing

We introduce our new technique that performs high-order rejection sampling with complexity independent from the size of the modulus q; the new complexity only depends on the size of the rejected interval. As depicted in Figure 2, the rejection zones in Dilithium are relatively narrow: by definition each coefficient z of $\mathbf{z}$ lies in the large interval $[-\gamma_1 - \beta, \gamma_1 + \beta]$ with $\gamma_1 = 2^{17}$ and $\beta = 78$ for Security Level 2, but rejection is necessitated only within the much smaller ranges of $[\gamma_1 - \beta, \gamma_1 + \beta]$ and $[-\gamma_1 - \beta, -\gamma_1 + \beta]$. The same holds for $\tilde{\mathbf{r}}$. Our new algorithm can leverage this small rejection range since it has complexity $\mathcal{O}(n^2 \cdot \log \beta n)$ instead of $\mathcal{O}(n^2 \cdot \log q)$ for previous works, with $\beta \ll q$ (see Table 3).

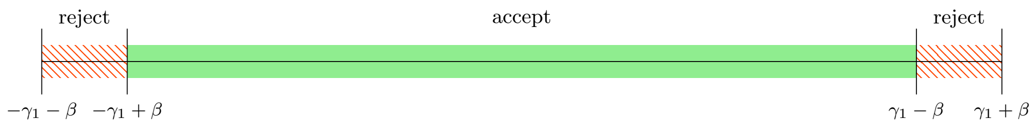

Figure 2. Illustration of the rejection sampling intervals for Dilithium.

Reduction to zero-testing. Our approach is as follows. For simplicity we first consider a single coefficient x. We start with arithmetic shares x_i of $x = x_1 + \cdots + x_n \bmod^{\pm} q$ such that $x \in [-b, b]$. The objective is to reject x if it falls outside the interval $]-a, a[$ for a given threshold $a < b$, all while ensuring no additional information about x is disclosed. Our approach is based on the following lemma, showing that integer comparison can be reduced to zero-testing modulo q. For a prime modulus q, such zero-testing can be high-order computed with complexity independent from the size of q, using the ZeroTestMult gadget from [CGMZ23], which we recall in Appendix B.1.

Lemma 2. *Let q be any positive integer. Let $a, b \in \mathbb{Z}$ with $0 < a < b < q/2$ and $b - a < 2^\rho$ for $\rho \in \mathbb{Z}$. Let $x \in [-b, b] \cap \mathbb{Z}$. We have:*

$$a \leq x \Leftrightarrow \left\lfloor \frac{x - a}{2^\rho} \right\rfloor = 0 \pmod q \tag{4}$$

Proof. If $a \leq x$, then $0 \leq x - a \leq b - a < 2^\rho$ and therefore $\lfloor (x - a)/2^\rho \rfloor = 0$. Conversely, if $\lfloor (x-a)/2^\rho \rfloor = 0 \pmod q$, then $\lfloor (x-a)/2^\rho \rfloor = \delta \cdot q$ for some $\delta \in \mathbb{Z}$, and therefore $x - a = 2^\rho \cdot \delta \cdot q + r$ for some $0 \leq r < 2^\rho$. We use $a + b < q$ and moreover $q \leq 2^\rho \cdot (q-1)$ for $q \geq 2$ and $\rho \geq 1$. This gives $x - a \geq -b - a > -2^\rho \cdot q + 2^\rho$, and therefore we get the inequality $\delta \cdot 2^\rho q + 2^\rho > x - a > -2^\rho q + 2^\rho$, which implies $\delta \geq 0$, which implies $x \geq a$. □

To perform the rejection sampling from the interval $]-a, a[$, it suffices to test if either $a \leq x$ or $a \leq -x$. The two tests cannot be performed using (4) separately, because in this case the adversary would learn on which side the coefficient x is out-of-bound, which would leak information about the secret-key. Instead, we simply high-order multiply the two results modulo q, so that if the coefficient is out of bound, the resultant product will be zero modulo q. This shows that the rejection sampling of x can be reduced to the zero-testing of the product below modulo q.

Corollary 1. *Let q be any prime integer. Let $a, b \in \mathbb{Z}$ with $0 < a < b < q/2$ and $b - a < 2^\rho$ for $\rho \in \mathbb{Z}$. Let $x \in [-b, b] \cap \mathbb{Z}$. We have:*

$$x \in]-a, a[\Leftrightarrow \left\lfloor \frac{x - a}{2^\rho} \right\rfloor \cdot \left\lfloor \frac{-x - a}{2^\rho} \right\rfloor \neq 0 \pmod q \tag{5}$$

Full procedure. The two arithmetic shifts by ρ bits in the above product can be computed thanks to the ShiftMod algorithm recalled in Appendix A. However, for this one needs to first obtain an arithmetic masking of x modulo $2^\rho q$ instead of modulo q only, where 2^ρ is an upper-bound on the size of the interval $[a, b]$. This can be achieved through a method akin to the one described in Section 3.2, which we describe in Appendix B.2. In summary, the rejection sampling of a single coefficient x proceeds as follows:

1. We first extend the arithmetic masking of x from modulo q to modulo $2^\rho q$, using a variant of the technique described in Section 3.2. We describe the corresponding LMSwitch algorithm in Appendix B.2.
2. We then compute the arithmetic shifts $\lfloor (x-a)/2^\rho \rfloor$ and $\lfloor (-x-a)/2^\rho \rfloor$ modulo q with the ShiftMod algorithm, recalled in Appendix A.
3. We high-order compute the product of the two above values modulo q, using SecMult; see [SPOG19] for the corresponding algorithm. We must also use a SNI mask refreshing algorithm [BBD+16].
4. Finally, we perform a zero-testing of the result, with complexity $\mathcal{O}(n^2)$, using the ZeroTest-Mult algorithm from [CGMZ23] recalled in Appendix B.1.

The full algorithm is provided in Algorithm 2. We show that the total complexity is $\mathcal{O}(n^2 \cdot (\rho + \log n))$ for n shares, which is independent from the modulus size. For Dilithium, one must take $a = \gamma_1 - \beta$ for the rejection sampling of $\mathbf{z}$, and $a = \gamma_2 - \beta$ for the rejection sampling of $\tilde{\mathbf{r}}$, with $\beta = 78$ for NIST Security Level 2. For the rejection sampling of $\mathbf{z}$, we have $b = \gamma_1 + \beta$, and $b = \gamma_2$ for the rejection sampling of $\tilde{\mathbf{r}}$. Since the parameter ρ must satisfy $2^\rho > b - a$, one must take $\rho = 8$ for the rejection sampling of $\mathbf{z}$, and $\rho = 7$ for the rejection sampling of $\tilde{\mathbf{r}}$.

Algorithm 2 RejectSampling

Input: A modulus q, an arithmetic sharing $x_1, \ldots, x_n$ of x such that $x_1 + \cdots + x_n = x \pmod q$, integers a, b, ρ such that $a < b < q/4$ with $b - a < 2^\rho$, and such that $|x| \leq b$.

Output: $u \in \{0,1\}$, with $u = 1$ if $|x| \geq a$, and $u = 0$ otherwise.

```
1: $u_1, \ldots, u_n \leftarrow \mathsf{LMSwitch}(q, \rho, (x_1, \ldots, x_n))$        $\triangleright\ x = u_1 + \cdots + u_n \pmod{2^\rho q}$
2: $(y_1, \ldots, y_n) \leftarrow (u_1 - a \bmod 2^\rho q, u_2, \ldots, u_n)$
3: $(y_1, \ldots, y_n) \leftarrow \mathsf{ShiftMod}_\rho(2^\rho \cdot q, (y_1, \ldots, y_n))$
4: $(y'_1, \ldots, y'_n) \leftarrow \mathsf{ResfreshMasks}(-u_1 - a \bmod 2^\rho q, -u_2, \ldots, -u_n)$
5: $(y'_1, \ldots, y'_n) \leftarrow \mathsf{ShiftMod}_\rho(2^\rho \cdot q, (y'_1, \ldots, y'_n))$
6: $(z_1, \cdots, z_n) \leftarrow \mathsf{SecMult}((y_1, \ldots, y_n), (y'_1, \ldots, y'_n))$
7: $b \leftarrow \mathsf{ZeroTestMult}_q(z_1, \ldots, z_n))$
8: return $b$
```

Complexity. The modulus switching at Line 1 has complexity $T_{\mathsf{LMSwitch}} = \alpha \cdot (2n^2 + 10n - 9) + 4n + 1 + 3n(n-1)/2$ where $\alpha = \lceil \log_2 n \rceil + 1$. The cost of the ShiftMod loops at lines 3 and 5 using [CGTZ23] is $\rho \cdot T_{\mathsf{ShiftMod}} = \rho \cdot (2n^2 + 10n - 9)$ operations. Furthermore, the number of operations of RefreshMasks and SecMult at line 6 are $3n \cdot (n-1)/2$ and $n \cdot (7n-5)/2$ respectively, and the ZeroTestMult gadget has complexity $\mathcal{O}(n^2)$. Therefore, asymptotically the number of operations of Algorithm 2 is $\mathcal{O}(n^2 \cdot (\alpha + \rho)) = \mathcal{O}(n^2 \cdot (\log n + \rho))$.

Security. The following theorem shows that our rejection sampling algorithm achieves the NIo property.

Theorem 4 **($(n-1)-$NIo of RejectSampling).** *For any set of t_1 probed variables, there exists a subset $I \subset [1, n]$, with $|I| \leq t_1$, of input indexes such that the t_1 probed variables can be perfectly simulated from $x_{|I}$ and the output b.*

Proof. The NIo property follows from the composition of the NI gadgets SecMult, ShiftMod, and LMSwitch, the SNI gadget RefreshMasks, and the NIo gadget ZeroTestMult.

4.3 Comparison with existing work

In Table 4 we provide a comparison with existing work [BBE+18] and [ABC+23]. We see that our technique is more efficient for all security orders. We also provide in Section 5.2 a comparison based on a concrete implementation on laptop computer and Cortex-M4.

	Security order t							
	2	3	4	5	6	7	8	9
[BBE+18]	8 566	15 682	26 141	38 313	52 868	69 136	89 417	111 411
[ABC+23]	2 902	5 354	9 180	13 531	18 756	24 506	31 979	39 977
This paper	834	1 362	2 078	2 832	3 687	4 643	5 943	7 149

Table 4. Operation count for rejection sampling of a single coefficient, with $q = 2^{23} - 2^{13} + 1$ and $\rho = 8$, with $n = t + 1$ shares.

4.4 On the commitment variable $\mathbf{w_1}$ in Dilithium

In previous works considering the masking of Dilithium [ABC+23,CGTZ23], the authors have argued that the commitment variable $\mathbf{w}_1$ generated in $(\mathbf{w}_1, \mathbf{w}_0) \leftarrow \mathsf{Decompose}_q(\mathbf{w})$ with $\mathbf{w} = \mathbf{Ay}$ need not be masked since it is also publicly computed during signature verification. Conveniently, this implies that the Keccak hash function H used for computing the challenge $\tilde{c} = H(M \| \mathbf{w}_1)$ need not be masked. However, this is only a heuristic assumption, as for aborted signatures, the knowledge of $\mathbf{w}_1$ might reveal additional information to an attacker.

In [CGTZ23], the authors claimed that this assumption has been analyzed rigorously in [DFPS23] and shown to hold unconditionally by providing an efficient simulator for all transcripts, including aborted ones; however, we argue that the argument from [DFPS23] does not apply directly to Dilithium. Namely, the authors of [DFPS23] considered a more generic lattice-based signature scheme where the commitment $\mathbf{w}$ is generated via $\mathbf{w} = \mathbf{Ay} \bmod q$, where $\mathbf{y} \leftarrow D$. Assuming that the distribution D has enough min-entropy, and that the matrix $\mathbf{A}$ has more columns than rows (so that the application $\mathbf{w} \rightarrow \mathbf{A} \cdot \mathbf{y}$ is highly surjective), the authors can apply the leftover hash lemma to argue that in failed transcripts, the commitment $\mathbf{w}$ can be simulated by a random vector modulo q. However, for Dilithium one cannot use directly the same argument, because in Dilithium the cyclotomic ring $R_q = \mathbb{Z}_q[X]/(X^{256} + 1)$ is isomorphic to the product of the rings $\mathbb{Z}_q[X]/(X - \zeta^i) \cong \mathbb{Z}_q$, over which the leftover hash lemma does not give a meaningful bound given the small dimension of the matrix $\mathbf{A}$. Therefore, even if the analysis from [DFPS23] can give us some confidence that in Dilithium revealing $\mathbf{w}_1$ from aborted signatures should be harmless, we do not have a rigorous proof yet. Therefore, to have a complete proof of security of a fully masked Dilithium with $\mathbf{w}_1$ computed in the clear (as we do in this paper) one should solve the following open problem, which is beyond the scope of this paper:

Open Problem 1: provide a security proof for Dilithium signatures, where the adversary additionally learns the commitments $\mathbf{w}_1$ from aborted signatures.

Additionally, for efficiency reason, the authors of [ABC+23,CGTZ23] consider an early abort strategy where if one of the (masked) coefficients of $\mathbf{z}$ or $\tilde{\mathbf{r}}$ is out of bound, the signature generation is immediately restarted. However, this implies that the adversary can learn not only the failed commitment $\mathbf{w}_1$, but also the position of some rejected coefficient in $\mathbf{z}$ or $\tilde{\mathbf{r}}$, which is not covered by the analysis in [DFPS23]. The advantage of the early abort approach is that it is roughly twice faster than performing the bound checking on *all* coefficients of $\mathbf{z}$ and $\tilde{\mathbf{r}}$ (see the next section). Intuitively, learning the position of the rejected coefficient should not help the adversary. Therefore, we consider the following additional open problem:

Open Problem 2: provide a security proof for Dilithium signatures, where the adversary additionally learns the commitments $\mathbf{w}_1$ from aborted signatures, and the position of the out-of-bound coefficients in $\mathbf{z}$ or $\tilde{\mathbf{r}}$, if any.

4.5 Rejection sampling of all coefficients

In this section, we consider the two possible strategies for the rejection sampling on $\mathbf{z}$ and $\tilde{\mathbf{r}}$: the early-abort strategy as in [ABC+23,CGTZ23], in which rejection sampling of each coefficient is performed separately, and the signature generation is immediately restarted if one coefficient is out-of-bound, and the no-early-abort strategy used in [BBE+18], in which rejection sampling is always performed on *all* coefficients of $\mathbf{z}$ and $\tilde{\mathbf{r}}$. We report on the implementation of these strategies in Section 5.2.

Early-abort strategy. The early-abort strategy is straightforward. We receive as input a vector of ℓ masked coefficients $x^{(j)} = x_1^{(j)} + \cdots + x_n^{(j)} \pmod q$ for $1 \leq j \leq \ell$, and we perform the rejection sampling iteratively. We provide a formal description of the algorithm in Appendix C.1, and we prove that the algorithm achieves the NIo security definition.

The advantage of the early-abort approach is that it is roughly twice faster than performing the bound checking on all coefficients of $\mathbf{z}$ and $\tilde{\mathbf{r}}$. However, as discussed in the previous section, getting a complete security proof for masked Dilithium will be more challenging, as for a failed signature transcript, the adversary additionally learns the position of at least one out-of-bound coefficient.

No-early-abort strategy. In the no-early-abort strategy, the rejection sampling is always performed on all coefficients, so that the adversary does not learn the position of the out-of-bound coefficient, if any. In [BBE+18], this is done as followed. The output bit of the rejection sampling for a single coefficient is kept in masked form, and a high-order secure And is performed to ensure that all output bits are 1. Therefore, if a coefficient must be rejected, the position of this coefficient is not revealed to the adversary, since this output bit is unmasked only at the end.

However, from our rejection sampling technique described in 4.2, we must proceed slightly differently, because the output bit of the zero-test is obtained in the clear, so one cannot perform a zero-test independently for each coefficient. Instead, we simply high-order multiply the results modulo q as in (5), and perform the zero-testing only once at the end. This corresponds to the following lemma. We describe the corresponding algorithm in Appendix C.2, and prove that it achieves the NIo security definition.

Lemma 3. *Let q be any prime integer. Let $a_j, b_j \in \mathbb{Z}$ with $0 < a_j < b_j < q/2$ and $b_j - a_j < 2^\rho$ for $\rho \in \mathbb{Z}$ and $\rho \geq 1$, for all $1 \leq j \leq \ell$. Let $x^{(j)} \in [-b_j, b_j] \cap \mathbb{Z}$ for $1 \leq j \leq \ell$. We have:*

$$\bigwedge_{j=1}^{\ell} \left(x^{(j)} \in]-a_j, a_j[\right) \Leftrightarrow \prod_{j=1}^{\ell} \left(\left\lfloor \frac{x^{(j)} - a_j}{2^\rho} \right\rfloor \cdot \left\lfloor \frac{-x^{(j)} - a_j}{2^\rho} \right\rfloor \right) \neq 0 \pmod q$$

5 Practical implementation

In this section, we describe our implementation of the new gadgets, building upon the publicly available code from [CGTZ23], to get a fully masked implementation of both Dilithium and ML-DSA. We clarify that our intention is to demonstrate and validate the proposed techniques, rather than to assert the security of the code against side-channel attacks as is. It is well known that one cannot really achieve a truly "leakage-free" execution from generic C code, especially when employing standard `gcc` compiler optimizations. Our timing reports for the Cortex M4 are based solely on this portable C code implementation. While developing a full-fledged low-level Dilithium implementation on a micro-controller to experimentally verify the absence of micro-architectural leakages would be a natural next step, this is out of scope of this work. The cycle counts for Cortex M4 were obtained by running the code on a `STM32F401RE` MCU embedded on a `STM32 Nucleo-64` developpement board. We compiled with the STM32Cube IDE in its default configuration. Furthermore, the computer used to get the laptop cycle counts is equipped with an `Intel(R) Core(TM) i7-1065G7 CPU` and the code was compiled with `O3` and `march=native` flags. The plain C code is publicly available at

https://github.com/fragerar/tches24_masked_Dilithium

5.1 Masked generation of the masking vector **y**

We have implemented our improved masked generation of **y** from Section 3.2 with the randomized version of Dilithium, targeting the widely-used Cortex-M4 platform, for two main reasons. Firstly, the risk of side-channel attacks is more pronounced on micro-controllers than on desktop or laptop computers. Secondly, our new gadget can work with 32-bit registers instead of 64-bit as the gadget from [CGTZ23]. Our new gadget is therefore particularly beneficial for 32-bit architectures like the Cortex-M4, where 64-bit operations are inherently less efficient than on modern computers with native 64-bit processing capabilities. We provide a comparison with [CGTZ23], which according to Table 2 is already significantly faster than [SPOG19] and [BBE+18] for small orders. The performance advantages of our method are evident in Table 5, where it demonstrates up to double the efficiency compared to the BtoAqExact algorithm from [CGTZ23].

B → A mod q	Security order t								
	1	2	3	4	5	6	7	8	9
[CGTZ23] BtoAqExact	6039	13110	24750	44967	81185	148296	276330	524856	1013893
BtoAqDelta	3682	7894	17645	31334	55232	99055	191557	353748	669662

Table 5. Cycle counts on Cortex-M4 for 18-bit Boolean to arithmetic modulo q conversion algorithms, up to security order $t = 9$, with $n = t + 1$ shares, for prime $q = 2^{23} - 2^{13} + 1$.

5.2 Masked rejection sampling

We have implemented our improved masked rejection sampling on both a laptop and a Cortex-M4 micro-controller. In Table 6, for the laptop implementation, we compare the single-coefficient performance of our new gadget with the masked rejection sampling gadget from [ABC+23]; we see that our new gadget is roughly 40% faster.

	Security order t								
	1	2	3	4	5	6	7	8	9
[ABC$^+$23]	1702	3604	6142	8585	11133	14414	18013	22512	27786
This paper	1208	2071	2942	4220	5669	7684	9867	12621	14383

Table 6. Cycle counts on laptop for rejection sampling of a single coefficient, with $q = 2^{23} - 2^{13} + 1$ and $\rho = 8$, with $n = t + 1$ shares.

This performance benefit extends to the full rejection sampling procedure. In Table 7, we report on the evaluation of both the no-early-abort and early-abort strategies (cf. Section 4.5) for our gadgets, alongside those from [ABC$^+$23]. Although the early-abort strategy is anticipated to double efficiency (a milestone nearly met by [ABC$^+$23] with a 46% time reduction), our implementation achieves a smaller efficiency gain (34% time reduction). This is attributed to the requirement for a zero-test gadget for each coefficient to assess early-abort possibility. Nevertheless, our early-abort strategy implementation outperforms [ABC$^+$23] by 41%. Comparatively, our gadget employing the more conservative no-early-abort strategy *matches* the efficiency of [ABC$^+$23]'s early-abort strategy.

		Security order t							
		1	2	3	4	5	6	7	8
No-early-abort	[ABC$^+$23]	3172	7070	11289	17795	24701	32294	40528	53119
	This paper	2376	3771	5698	8531	11471	15130	19238	24934
Early-abort	[ABC$^+$23]	1800	3804	6389	9549	13244	16648	20938	27112
	This paper	1420	2231	3232	4817	7209	8550	10717	13975

Table 7. Cycle counts on laptop for the rejection sampling of all coefficients in $\mathbf{z}$ and $\tilde{\mathbf{r}}$ (in thousands of cycles).

In addition to the laptop implementation, we report on the performance of our technique on Cortex M4 in Table 8. Our method offers a more pronounced improvement over [ABC$^+$23] compared to the laptop results, at least for a non-bitsliced implementation. Namely, in their own implementation, the authors of [ABC$^+$23] use the very efficient bitsliced implementation from [BC22], whereas our algorithms are not compatible with bitslicing. For a more comprehensive comparison, these evaluations should ideally be conducted using optimized ARM assembly code rather than portable C code.

	Security order t								
	1	2	3	4	5	6	7	8	9
[ABC$^+$23]	46206	118065	205641	335730	481501	653039	840302	1080287	1336004
This paper	20728	40542	65533	100704	139311	184026	235005	303972	369945

Table 8. Cycle counts on Cortex-M4 for rejection sampling of a single coefficient, with $q = 2^{23} - 2^{13} + 1$ and $\rho = 8$, with $n = t + 1$ shares.

5.3 Fully masked Dilithium signature scheme

Desktop implementation. Table 9 provides a detailed breakdown of the various components involved in the signature process, for fully-masked randomized Dilithium signatures. It shows that although the rejection sampling procedure remains a significant factor, especially for lower orders, its impact is lessened compared to previous works [ABC+23,CGTZ23]. The primary bottleneck has shifted to the Decompose procedure, for which we simply reused the code from [CGTZ23].

	Security order t						
	0	1	2	3	4	5	6
NTTs	-	135	214	282	346	369	476
Sample y	-	900	1480	2959	4608	7170	11921
Compute Ay	-	217	296	391	479	613	667
Decompose	-	7922	18700	30409	47479	62969	84578
$z = y + c \cdot s_1$	-	158	223	291	362	446	508
Reject	-	6235	9913	14424	21440	28601	38115
$w - c \cdot s_2$	-	85	118	159	193	237	273
Dilithium2	483	15812(×33)	31124(×64)	49124(×102)	75138(×156)	100676(×208)	136825(×283)

Table 9. Cycle counts on laptop for the full randomized Dilithium (Level 2) with penalty factor, including the main operations within Dilithium, using the early-abort strategy. Average over 500 executions of the signature at each order (in thousands of cycles).

Comparison with [CGTZ23]. In Table 10, we provide a comparison with [CGTZ23], for a laptop implementation. We see that for security orders $t \geq 2$, the new implementation is roughly 25% faster.

	Security order t						
	0	1	2	3	4	5	6
Dilithium2 [CGTZ23]	483	16636(×34)	41932(×87)	67644(×140)	98257(×203)	137418(×285)	171057(×354)
Dilithium2	483	15812(×33)	31124(×64)	49124(×102)	75138(×156)	100676(×208)	136825(×283)

Table 10. Cycle counts on laptop for the full randomized Dilithium (Level 2) with penalty factor, in [CGTZ23] and in this paper. Average over 500 executions of the signature at each order (in thousands of cycles).

Implementation on Cortex M4. In Table 11, we provide the runtime of one main loop iteration of fully-masked randomized Dilithium on Cortex-M4. Since the MCU we used for the experiments is quite limited in terms of memory (less than 100kB) and since the reference code of Dilithium is not memory optimized[1], the high-order masked signature would not fit in memory without further optimizations. Thus we provide an estimation of the total runtime by running

[1] According to the pqm4 library, the unmasked reference implementation is already using more than half the memory of our MCU.

every (main) component independently at every order and then summing their runtime. The values in the table correspond to one iteration of the rejection loop, this means that they must be multiplied by a factor of 4.25 to estimate the average total execution time of the signature. The total runtime for the unmasked signature has been obtained by running the full signature and then dividing by 4.25.

	Security order t						
	0	1	2	3	4	5	6
Sample y	-	4424	8977	19183	33393	58152	103232
Compute Ay	-	3343	5014	6686	8357	10029	11700
Decompose	-	48108	126401	221987	362367	520006	705363
Reject	-	40843	79748	128631	197879	273420	361180
Dilithium2 (one iter.)	≈ 7400	≈ 96718	≈ 220140	≈ 376487	≈ 601996	≈ 861607	≈ 1181475

Table 11. Cycle counts estimation on Cortex-M4 for one iteration of the main loop of Dilithium (security level 2, in thousands of cycles). The last line should be multiplied by a factor of 4.25 to get an estimation of the total runtime of the signature.

5.4 Implementation of ML-DSA

In this section, we describe our fully masked implementation of ML-DSA. The primary distinction between Dilithium (Version 3.1) [BDK+21] and the ML-DSA draft standard [NIS23] lies in the derivation of the seed ρ' for generating the vector $\mathbf{y}$. In Dilithium's randomized signatures, ρ' is obtained directly as a 512-bit random string. In contrast, ML-DSA employs a "hedged" mechanism for ρ''s generation, deriving it pseudo-randomly from the signer's private-key, the message M, and a random 256-bit string *rnd*. For ML-DSA's deterministic variant, *rnd* is replaced with a fixed 256-bit string. These procedural variations between Dilithium and ML-DSA are detailed in Table 12.

	Deterministic	Randomized
Dilithium	$\rho' = H(K \,\|\, \mu)$	$\rho' \leftarrow \{0,1\}^{512}$
ML-DSA	$rnd \leftarrow \{0\}^{256}$ $\rho' = H(K \,\|\, rnd \,\|\, \mu)$	$rnd \leftarrow \{0,1\}^{256}$ $\rho' = H(K \,\|\, rnd \,\|\, \mu)$

Table 12. Generation of ρ' from $\mu = H(tr \| M)$ in Dilithium vs ML-DSA, for deterministic vs randomized signatures. The vector $\mathbf{y}$ is then generated as $\mathbf{y} \leftarrow \mathsf{ExpandMask}(\rho', \kappa)$.

From a security perspective, the difference between ML-DSA and Dilithium is that for randomized signatures, ML-DSA remains secure even if the output of the random number generator is fixed or predictable to the adversary (thanks to the "hedged" procedure), whereas Dilithium would be insecure in that case. However, a drawback of ML-DSA compared to Dilithium is that the sampling of $\mathbf{y}$ cannot be pre-computed off-line anymore.

As discussed in Section 3.1, in the randomized variant of Dilithium, each coefficient y within the vector $\mathbf{y}$ is computationally indistinguishable from the uniform distribution in the interval

$]-\gamma_1, \gamma_1]$. In masking Dilithium, we adopted the strategy utilized in [ABC+23,CGTZ23] by directly generating each masked coefficient y uniformly within the same interval, bypassing the ExpandMask function. In principle one could proceed similarly for ML-DSA, since in ML-DSA the coefficients y are also computationally indistinguishable from uniform in $]-\gamma_1, \gamma_1]$. However, such a method would negate the "hedged" characteristic of $\mathbf{y}$'s generation in ML-DSA, which upholds its security even under RNG compromise by an adversary.

Therefore, for both deterministic and randomized versions of ML-DSA, we adopt a similar approach to deterministic Dilithium by masking the generation of $\rho' = H(K\|H(tr\|M))$ and obtain Boolean shares ρ'_i of the seed ρ'; the ExpandMask function is also masked in this process. We utilized the masked Keccak implementation from [CGMZ22] for this purpose. Experimental results showcased in Table 13 aim to evaluate the performance impact of employing a masked hash function. As anticipated, the performance hit is significant.[2] Table 14 presents the performance metrics of our fully masked ML-DSA implementation, showing a 29% increase in execution time relative to randomized Dilithium at order 2.[3]

	Security order t							
	1	2	3	4	5	6	7	8
Randomized Dilithium	211	321	662	1148	1761	2826	5230	9074
Randomized ML-DSA	984	2782	5081	9140	13676	19696	26023	36024

Table 13. Cycle counts on laptop for the generation of the $\mathbf{y}$ vector in randomized Dilithium and ML-DSA, with $n = t + 1$ shares (in thousands of cycles).

	Security order t						
	0	1	2	3	4	5	6
NTTs	-	144	212	288	342	379	506
Sample y	-	4426	11765	22422	37449	58319	85859
Compute Ay	-	261	288	404	467	641	702
Decompose	-	7736	17969	31183	45084	65397	87996
$z = y + c \cdot s_1$	-	154	215	301	354	464	529
Reject	-	6192	9746	14977	21375	30098	39375
$w - c \cdot s_2$	-	84	117	164	192	256	283
ML-DSA	483	19195(×40)	40601(×84)	70160(×145)	105877(×219)	156399(×324)	216357(×448)

Table 14. Cycle counts on laptop for the full randomized ML-DSA with penalty factor, including the main operations within ML-DSA, using the early-abort strategy. Average over 500 executions of the signature at each order (in thousands of cycles).

[2] Note that the larger figures for the sampling of y in Table 9 compared to the first line of Table 13 are attributable to the signature procedure necessitating roughly 4 repetitions on average.

[3] Our ML-DSA version builds upon the masked Dilithium implementation, incorporating masked Keccak for generating $\mathbf{y}$ as outlined in Table 12. While we do not claim a fully compliant ML-DSA implementation, the performance metrics are expected to be closely comparable.

Finally, to preserve the "hedged" feature in the randomized ML-DSA but with unmasked H and $\mathsf{ExpandMask}$, one possibility (left as future work) would be to keep the standard $\rho' = H(K \,\|\, rnd \,\|\, \mu)$ and $\mathbf{y} \leftarrow \mathsf{ExpandMask}(\rho', \kappa)$ procedure of ML-DSA, but initially unmasked. Recall that in $\mathsf{ExpandMask}$, each coefficient y is generated by first computing $y' \in \{0, \ldots, 2\gamma_1 - 1\}$ and then $y = \gamma_1 - y'$. Therefore, to get high-order security, one could generate as in Section 3.1 n random Boolean shares u_i and apply the Boolean to arithmetic conversion to the shares of $y'' = (y' \oplus u_1) \oplus u_2 \oplus \cdots \oplus u_n$, and eventually compute $y = \gamma_1 - y''$ over the arithmetic shares of y''. This approach would retain the "hedged" protection against a 0-probe adversary controlling the RNG, while also being provably secure against up to $n-1$ probes, and offering performance on par with fully masked randomized Dilithium (as detailed in Table 9).

6 Conclusion

We have described advanced high-order masking techniques for the Dilithium signature scheme, with better computational efficiency. This includes a refined Boolean-to-arithmetic conversion algorithm, and a novel rejection sampling method with complexity independent of the modulus size. Our new gadgets are proven secure in the classical ISW probing model. Finally, we have also described a fully masked implementation of both Dilithium and ML-DSA incorporating these new gadgets.

Acknowledgements. The first and second authors were supported by the ERC Advanced Grant no. 787390.

References

ABC+23. Melissa Azouaoui, Olivier Bronchain, Gaëtan Cassiers, Clément Hoffmann, Yulia Kuzovkova, Joost Renes, Tobias Schneider, Markus Schönauer, François-Xavier Standaert, and Christine van Vredendaal. Protecting dilithium against leakage revisited sensitivity analysis and improved implementations. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2023(4):58–79, 2023. https://eprint.iacr.org/2022/1406.

BBD+15. Gilles Barthe, Sonia Belaïd, François Dupressoir, Pierre-Alain Fouque, Benjamin Grégoire, and Pierre-Yves Strub. Verified proofs of higher-order masking. In *Advances in Cryptology - EUROCRYPT 2015 - Proceedings, Part I*, pages 457–485, 2015.

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.

BC22. Olivier Bronchain and Gaëtan Cassiers. Bitslicing arithmetic/boolean masking conversions for fun and profit with application to lattice-based kems. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2022(4):553–588, 2022. https://ia.cr/2022/158.

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

BDE+18. Jonathan Bootle, Claire Delaplace, Thomas Espitau, Pierre-Alain Fouque, and Mehdi Tibouchi. LWE without modular reduction and improved side-channel attacks against BLISS. In Thomas Peyrin and Steven D. Galbraith, editors, *Advances in Cryptology - ASIACRYPT 2018 - Proceedings, Part I*, volume 11272 of *Lecture Notes in Computer Science*, pages 494–524. Springer, 2018.

BDK$^+$21. Shi Bai, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, Peter Schwabe, Gregor Seiler, and Damien Stehlé. Crystals-dilithium algorithm specifications and supporting documentation (version 3.1), 2021. https://pq-crystals.org/dilithium/data/dilithium-specification-round3-20210208.pdf.

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.

CGMZ23. Jean-Sébastien Coron, François Gérard, Simon Montoya, and Rina Zeitoun. High-order polynomial comparison and masking lattice-based encryption. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2023(1):153–192, 2023. https://ia.cr/2021/1615.

CGTZ23. Jean-Sébastien Coron, François Gérard, Matthias Trannoy, and Rina Zeitoun. Improved gadgets for the high-order masking of dilithium. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2023(4):110–145, 2023. https://eprint.iacr.org/2023/896.

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.

DFPS23. Julien Devevey, Pouria Fallahpour, Alain Passelègue, and Damien Stehlé. A detailed analysis of fiat-shamir with aborts. Cryptology ePrint Archive, Paper 2023/245, 2023. https://eprint.iacr.org/2023/245.

GLP12. Tim Güneysu, Vadim Lyubashevsky, and Thomas Pöppelmann. Practical lattice-based cryptography: A signature scheme for embedded systems. In Emmanuel Prouff and Patrick Schaumont, editors, *Cryptographic Hardware and Embedded Systems – CHES 2012*, pages 530–547, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

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

Lyu09. Vadim Lyubashevsky. Fiat-shamir with aborts: Applications to lattice and factoring-based signatures. In *Advances in Cryptology–ASIACRYPT 2009: 15th International Conference on the Theory and Application of Cryptology and Information Security, Tokyo, Japan, December 6-10, 2009. Proceedings 15*, pages 598–616. Springer, 2009.

LZS$^+$21. Yuejun Liu, Yongbin Zhou, Shuo Sun, Tianyu Wang, Rui Zhang, and Jingdian Ming. On the security of lattice-based fiat-shamir signatures in the presence of randomness leakage. *IEEE Trans. Inf. Forensics Secur.*, 16:1868–1879, 2021.

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

MUTS22. Soundes Marzougui, Vincent Ulitzsch, Mehdi Tibouchi, and Jean-Pierre Seifert. Profiling side-channel attacks on dilithium: A small bit-fiddling leak breaks it all. Cryptology ePrint Archive, Paper 2022/106, 2022. https://eprint.iacr.org/2022/106.

NIS23. NIST. Module-lattice-based digital signature standard. (department of commerce, washington, d.c.), federal information processing standards publication (fips) nist fips 204 ipd., 2023. https://doi.org/10.6028/NIST.FIPS.204.ipd.

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.

A The arithmetic shift algorithm ShiftMod

We recall the arithmetic shift algorithm ShiftMod from [CGTZ23]. The basic algorithm takes as input an arithmetic sharing of x modulo $2q$, and outputs an arithmetic sharing of $\lfloor x/2 \rfloor$ modulo q, without leaking information about x. The algorithm relies on the 1-bit Boolean to arithmetic masking algorithm from [SPOG19]. More precisely, the ShiftMod gadget high-order computes the shift:

$$\left\lfloor \frac{x}{2} \right\rfloor = \frac{x - (x \bmod 2)}{2} \pmod{q}$$

For this, one first performs a 1-bit Boolean to arithmetic modulo $2q$ conversion of the bit $(x \bmod 2)$, using the 1bitB2A gadget from [SPOG19]. By subtracting the result to x, one can obtain an arithmetic sharing of $x - (x \bmod 2)$ where all shares modulo $2q$ are even, so one can divide by 2 all shares independently and obtain an arithmetic sharing of $\lfloor x/2 \rfloor$ modulo q; see Alg. 3 below. By iterating the above ShiftMod algorithm β times, one obtains the $\mathsf{ShiftMod}_\beta$ algorithm (see Alg. 4).

Algorithm 3 ShiftMod

```
Input: A modulus $q' = 2q$ and $x_1, \ldots, x_n \in \mathbb{Z}_{2q}$
Output: $a_1, \ldots, a_n \in \mathbb{Z}_q$ such that $a_1 + \cdots + a_n = \lfloor (x_1 + \cdots + x_n)/2 \rfloor \pmod q$
1: for $i = 1$ to $n$ do $b_i \leftarrow x_i \,\&\, 1$
2: $(y_1, \ldots, y_n) \leftarrow \mathsf{1bitB2A}(2q, (b_1, \ldots, b_n))$
3: for $i = 1$ to $n$ do $z_i \leftarrow x_i - y_i \bmod 2q$
4: for $i = 1$ to $n - 1$ do
5:     $z_n \leftarrow z_n + (z_i \,\&\, 1) \bmod 2q$
6:     $z_i \leftarrow z_i - (z_i \,\&\, 1) \bmod 2q$
7: end for
8: for $i = 1$ to $n$ do $a_i \leftarrow z_i \gg 1$
9: return $a_1, \ldots, a_n$
```

Algorithm 4 $\mathsf{ShiftMod}_\beta$

```
Input: A modulus $q' = q \cdot 2^\beta$, and $x_1, \ldots, x_n \in \mathbb{Z}_{q2^\beta}$
Output: $a_1, \ldots, a_n \in \mathbb{Z}_q$ such that $a_1 + \cdots + a_n = \lfloor (x_1 + \cdots + x_n)/2^\beta \rfloor \pmod q$
1: for $i = 0$ to $\beta - 1$ do $(x_1, \ldots, x_n) \leftarrow \mathsf{ShiftMod}(2^{\beta - i} q, (x_1, \ldots, x_n))$
2: return $x_1, \ldots, x_n$
```

From [CGTZ23], the complexity of $\mathsf{ShiftMod}_\beta$ is $T_{\mathsf{ShiftMod}_\beta}(n) = \beta \cdot (2n^2 + 10n - 9)$, assuming that operations modulo q have unit cost. The authors of [CGTZ23] prove the NI property of ShiftMod, based on the free-SNI of 1bitB2A from [SPOG19], which is also proven in [CGTZ23]. By composition, we get the NI property of $\mathsf{ShiftMod}_\beta$.

Theorem 5 ([CGTZ23]). *The* $\mathsf{ShiftMod}_\beta$ *algorithm achieves the* NI *property.*

B Algorithms for rejection sampling

B.1 Zero-testing algorithm

We recall the ZeroTestMult algorithm from [CGMZ23], based on converting from arithmetic masking modulo q to multiplicative masking, for a prime modulus q. At the end of the algorithm, one obtains multiplicative shares u_i such that $u_1 \cdots u_n \cdot x = B \pmod q$ for a known B. Therefore, if the secret value x is 0, then $B = 0$, whereas for any $x \neq 0$, the variable B is random in $\mathbb{Z}_q^*$. This enables to distinguish the two cases, without leaking more information about x. The complexity is $\mathcal{O}(n^2)$, independent from the size of q. The authors proved that it achieves the NIo property.

Algorithm 5 ZeroTestMult

```
Input: $x_1, \ldots, x_n \in \mathbb{Z}_q$ for prime $q$.
Output: $b \in \{0,1\}$ with $b = 1$ if $\sum_i x_i = 0 \pmod q$ and $b = 0$ otherwise
1: $(B_1, \ldots, B_n) \leftarrow (x_1, \ldots, x_n)$
2: for $j = 1$ to $n$ do
3:     $u_j \leftarrow \mathbb{Z}_q^*$
4:     $(B_1, \ldots, B_n) \leftarrow (u_j \cdot B_1 \bmod q, \ldots, u_j \cdot B_n \bmod q)$
5:     $(B_1, \ldots, B_n) \leftarrow \mathsf{LinearRefreshMasks}(B_1, \ldots, B_n)$
6: end for
7: $B \leftarrow B_1 + \cdots + B_n \bmod q$
8: if $B = 0$ then return 1 else return 0
```

B.2 Arithmetic masking: switching to a larger modulus

Given as input an arithmetic sharing of $x = x_1 + \cdots + x_n \bmod^{\pm} q$, we show how to obtain an arithmetic sharing of x modulo a larger modulus $2^\rho q$, for any $\rho > 0$. The technique works under the condition that $|x| < q/4$, which is the case in Dilithium. We fix $\alpha := \lceil \log_2 n \rceil + 1$, where n is the number of shares.

For this, as in Section 3.2, we write $x_1 + \cdots + x_n = x + \delta \cdot q$ for some $\delta \in \mathbb{Z}$, and we show how to derive an arithmetic masking of δ modulo 2^ρ. As previously, we write the previous equation as an arithmetic sharing of $z := x + \delta \cdot q$ modulo $2^\rho q$:

$$z = x_1 + \cdots + x_n = x + \delta \cdot q \pmod{2^\rho q} \tag{6}$$

As in Section 3.2, we apply the "Modulus Switching Lemma" (Lem. 2) with $p_1 := 2^\rho q$ and $p_2 := 2^{\rho+\alpha}$ to the shares x_i of z in (6), which gives an arithmetic sharing of $\lfloor z \cdot p_2/p_1 \rfloor = \lfloor z2^\alpha/q \rfloor$, up to some small additive error $0 \le e < n$. More precisely, from (6) we get arithmetic shares y_i satisfying:

$$y_1 + \cdots + y_n = \left\lfloor \frac{z2^\alpha}{q} \right\rfloor + e = \left\lfloor \frac{x2^\alpha}{q} \right\rfloor + e + \delta \cdot 2^\alpha \pmod{2^{\rho+\alpha}}$$

From $|x| < q/4$, we have $0 < x + q/4 < q/2$ and therefore $0 \le \lfloor x2^\alpha/q \rfloor + 2^{\alpha-2} < 2^{\alpha-1}$. Using $0 \le e < n \le 2^{\alpha-1}$, we get $0 \le \lfloor x \cdot 2^\alpha/q \rfloor + 2^{\alpha-2} + e < 2^\alpha$. Therefore, by applying the $\mathsf{ShiftMod}_\alpha$ arithmetic shift by α bits on the shares y_i, with $2^{\alpha-2}$ as an additive factor on y_1, we can get rid of the term $\lfloor x \cdot 2^\alpha/q \rfloor + e$ and obtain an arithmetic sharing of δ modulo 2^ρ:

$$\delta_1 + \cdots + \delta_n = \left\lfloor \frac{y_1 + \cdots + y_n + 2^{\alpha-2}}{2^\alpha} \right\rfloor = \delta \pmod{2^\rho}$$

Eventually, from (6) we obtain an arithmetic masking of x modulo $2^\rho q$, as required:

$$x = \sum_{i=1}^{n} x_i - \delta \cdot q = \sum_{i=1}^{n} (x_i - q \cdot \delta_i) \pmod{2^\rho q}$$

Algorithm 6 Larger Modulus Switching ($\mathsf{LMSwitch}$)

```
Input: A modulus $q$, a parameter $\rho$, an arithmetic sharing $x_1, \dots, x_n$ of $x = x_1 + \cdots + x_n \bmod^{\pm} q$,
    with $|x| < q/4$.
Output: An arithmetic sharing $v_1, \dots, v_n$ such that $v_1 + \cdots + v_n = x \pmod{2^\rho q}$.
1: $\alpha \leftarrow \lceil \log_2 n \rceil + 1$
2: for $i = 1$ to $n$ do $y_i \leftarrow \lfloor x_i \cdot 2^\alpha / q \rfloor$
3: $y_1 \leftarrow y_1 + n - 1 + 2^{\alpha-2} \bmod 2^{\rho+\alpha}$
4: $(\delta_1, \dots, \delta_n) \leftarrow \mathsf{ShiftMod}_\alpha(2^{\rho+\alpha}, (y_1, \dots, y_n))$
5: $(\delta_1, \dots, \delta_n) \leftarrow \mathsf{RefreshMasks}(\delta_1, \dots, \delta_n)$
6: for $i = 1$ to $n$ do $v_i \leftarrow x_i - q \cdot \delta_i \bmod 2^\rho q$
7: return $(v_1, \dots, v_n)$
```

Complexity. The cost of the $\mathsf{ShiftMod}_\alpha$ at Line 4 using [CGTZ23] is $\alpha \cdot T_{\mathsf{ShiftMod}} = \alpha \cdot (2n^2 + 10n - 9)$ operations, with $\alpha = \lceil \log_2 n \rceil + 1$. Furthermore, the For loops at lines 2 and 6 both cost $2n$ operations, and $\mathsf{RefreshMasks}$ costs $3n(n-1)/2$ operations. Therefore the overall complexity is $T_{\mathsf{LMSwitch}} = \alpha \cdot (2n^2 + 10n - 9) + 4n + 1 + 3n(n-1)/2$ and asymptotically the number of operations is $\mathcal{O}(\alpha \cdot n^2) = \mathcal{O}(n^2 \log n)$.

Theorem 6 (($(n-1)-\mathsf{NI}$ of $\mathsf{LMSwitch}$)**.** *For any set of t_1 probed variables, there exists a subset $I \subset [1, n]$, with $|I| \le t_1$, of input indexes such that the t_1 probed variables can be perfectly simulated from $x_{|I}$*

Proof. The NI property follows from the composition of the NI gadget $\mathsf{ShiftMod}_\alpha$ with the SNI gadget $\mathsf{RefreshMasks}$.

C Strategies for rejection sampling

C.1 Rejection sampling: early-abort

Note that our algorithm also takes as input an independent bound a_j for each $x^{(j)}$. This is because in Dilithium, one must use a different bound for the rejection sampling of $\mathbf{z}$ and $\tilde{\mathbf{r}}$, namely $a_j = \gamma_1 - \beta$ for $\mathbf{z}$, and $a_j = \gamma_2 - \beta$ for $\tilde{\mathbf{r}}$.

Algorithm 7 Rejection sampling: early-abort

```
Input: A sequence of $\ell$ arithmetic shared mod $q$ values $\{(x_1^{(j)}, \dots, x_n^{(j)})\}_{1 \le j \le \ell}$ such that $|x^{(j)}| \le$
    $b_j$ where $x^{(j)} = x_1^{(j)} + \cdots + x_n^{(j)} \bmod^{\pm} q$, and bounds $a_j$ for $1 \le j \le \ell$.
Output: A bit $u$ such that $u = 0$ if all coefficients satisfy $|x^{(j)}| < a_j$ and $u = 1$ otherwise.
1: for $j = 1$ to $\ell$ do
2:     $u_j \leftarrow \mathsf{RejectSampling}(q, (x_1^{(j)}, \dots, x_n^{(j)}), a_j, b_j, \rho)$
3:     if $u_j = 1$ then return 1
4: end for
5: return 0
```

Lemma 4 (NIo **security**). *Any t probes in the above algorithm can be perfectly simulated from the knowledge of all $x^{(j)}_{|I}$ for $1 \leq j \leq \ell$ for $|I| \leq t$, and the position of the first out-of-bound coefficient $x^{(j)}$, if any.*

Proof. The proof follows from the NIo security of RejectSampling, for which the output u_j can be given to the simulator since we are given the position of the first out-of-bound coefficient $x^{(j)}$, if any.

C.2 No-early-abort strategy

We describe the rejection sampling algorithm with the no-early-abort strategy. We denote by RejectSamplingShares the RejectSampling algorithm (see Alg. 2) where the shares $z_1, \ldots, z_n$ are output directly after Line 6.

Algorithm 8 Rejection sampling: no-early-abort

Input: A sequence of ℓ arithmetic shared mod q values $\{(x_1^{(j)}, \ldots, x_n^{(j)})\}_{1 \leq j \leq \ell}$ such that $|x^{(j)}| \leq b_j$ where $x^{(j)} = x_1^{(j)} + \cdots + x_n^{(j)} \bmod^{\pm} q$, and bounds a_j for $1 \leq j \leq \ell$.
Output: A bit u such that $u = 0$ if all coefficients satisfy $|x^{(j)}| < a_j$ and $u = 1$ otherwise.

```
1: $(u_1^{(0)}, \ldots, u_n^{(0)}) \leftarrow (1, 0, \ldots, 0)$
2: for $j = 1$ to $\ell$ do
3:     $u_1^{(j)}, \ldots, u_n^{(j)} \leftarrow \mathsf{RejectSamplingShares}(q, (x_1^{(j)}, \ldots, x_n^{(j)}), a_j, b_j, \rho)$
4:     $u_1^{(j)}, \ldots, u_n^{(j)} \leftarrow \mathsf{SecMult}((u_1^{(j-1)}, \ldots, u_n^{(j-1)}), (u_1^{(j)}, \ldots, u_n^{(j)}))$
5: end for
6: return $\mathsf{ZeroTest}_q(u_1^{(\ell)}, \ldots, u_n^{(\ell)})$
```

Lemma 5 (NIo **security**). *Any t probes in the above algorithm can be perfectly simulated from the knowledge of all $x^{(j)}_{|I}$ for $1 \leq j \leq \ell$ for $|I| \leq t$, and knowing the output bit b of the rejection sampling.*

Proof. The NIo security follows from the composition of the NIo gadget ZeroTest, the NI security of RejectSamplingShares and the SNI security of the SecMult.