Improved Gadgets for the High-Order Masking of Dilithium

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

[1] University of 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. We present novel and improved high-order masking gadgets for Dilithium, a post-quantum signature scheme that has been standardized by the National Institute of Standards and Technologies (NIST). Our proposed gadgets include the ShiftMod gadget, which is used for efficient arithmetic shifts and serves as a component in other masking gadgets. Additionally, we propose a new algorithm for Boolean-to-arithmetic masking conversion of a μ-bit integer x modulo any integer q, with a complexity that is independent of both μ and q. This algorithm is used in Dilithium to mask the generation of the random variable y modulo q. Moreover, we describe improved techniques for masking the Decompose function in Dilithium. Our new gadgets are proven to be secure in the t-probing model.
We demonstrate the effectiveness of our countermeasures by presenting a complete high-order masked implementation of Dilithium that utilizes the improved gadgets described above. We provide practical results obtained from a C implementation and compare the performance improvements provided by our new gadgets with those of previous work.

Keywords: Lattice-based signature, Dilithium, high-order masking

1 Introduction

Dilithium signatures. The impending development of scalable quantum computers threatens the security of prevailing asymmetric cryptography primitives. In 1994, Shor showed that both RSA and ECC can be broken by a quantum computer in polynomial time. As a result, the National Institute of Standards and Technologies (NIST) launched a competition in 2016 to select new standards for digital signature and public-key encryption/key-establishment algorithms. The competition came to a conclusion in 2022, with the announcement of CRYSTALS-Kyber for public-key encryption and CRYSTALS-Dilithium, FALCON, and SPHINCS+ for digital signature algorithms. This paper will focus on Dilithium, the primary post-quantum signature scheme standardized by NIST.

Dilithium is a signature scheme based on lattice cryptography [BDK+21]. It utilizes the "Fiat-Shamir with Aborts" technique developed by Lyubashevsky [Lyu09], which is based on rejection sampling. The security of Dilithium relies on the computational hardness of finding short vectors in lattices. NIST selected Dilithium as the primary post-quantum signature scheme due to its excellent performance and small signature size. To make implementation easier, Dilithium employs uniform sampling, as in [GLP12], rather than discrete Gaussian sampling, which would be much harder to protect against side-channel attacks.

Side-channel vulnerabilities. Like most implementations of cryptographic primitives, the Dilithium signature scheme is vulnerable to side-channel attacks. These attacks focus on exploiting physical leakage that occurs during the execution of an algorithm when run on an embedded device. Recently, Liu *et al.* [LZS+21] described a side-channel attack on Dilithium that uses only a single bit of the nonce y for multiple signatures. They reduced the problem of recovering the secret key to a variant of the Integer Learning with Errors (ILWE) problem [BDE+18] and proved that it can be solved using least squares method in polynomial time. Similarly, the authors of [MUTS22] used a profiled attack based on machine learning, targeting the secret signing nonce y for multiple signatures, followed by least-square regression and integer programming to solve a noisy linear system and recover the secret key.

The masking countermeasure. Side-channel attacks can be prevented by ensuring that any computation, and consequently any leakage, is independent of any secret the algorithm wields. At the simplest level, this can be achieved by splitting each secret variable x into shares $x = x_1 \oplus x_2$, where x_1 is a uniform random mask and $x_2 = x \oplus x_1$. By processing each share independently without recombining them, an attacker who probes only one variable through side-channel attacks will learn nothing about the secret key, since both x_1 and x_2 are uniformly distributed. More generally, [ISW03] introduced the t-probing model, where an attacker can probe up to t variables. The authors showed how to protect implementations using masks with $n = 2t+1$ shares, i.e., $x = x_1 \oplus \cdots \oplus x_n$. More precisely, they showed how any Boolean circuit C of size $|C|$ can be transformed into an equivalent circuit $\tilde{C}$ that is t-probing secure with size $\mathcal{O}(n^2|C|)$. The number of shares was later reduced to $n = t + 1$ in [BBD+15]. The authors introduced the (Strong) Non-Interference (SNI and NI) security notions and composability theorems so that the overall probing security of a scheme can be broken down to the security of its elementary parts.

Although the generic conversion algorithm from [ISW03] can be used to make Dilithium signature generation secure against side-channel attacks, the specific operations used in lattice-based cryptography make this approach inefficient in practice. This is because lattice-based cryptography uses both arithmetic and Boolean operations, and arithmetic masking ($x = x_1 + \cdots + x_n \bmod q$) is preferred for arithmetic computations, while Boolean masking ($x = x_1 \oplus \cdots \oplus x_n$) is used for Boolean operations, requiring frequent conversions between the two masked representations. To address this issue, efficient high-order conversion algorithms have been developed to work with any modulus q. The first high-order conversion between arithmetic and Boolean masking was described in [CGV14] for a power-of-two modulus q, for masking block-ciphers and hash functions that combine Boolean and arithmetic operations, such as SHA-1. The conversion technique from [CGV14] was then extended to any modulus q in [BBE+18]. Additionally, [SPOG19] presented an efficient 1-bit Boolean to arithmetic conversion modulo q with a complexity of $\mathcal{O}(n^2)$. A generic conversion algorithm between arithmetic and Boolean masking was proposed in [CGMZ22], based on the randomized table countermeasure from [Cor14]. Furthermore, dedicated gadgets have been developed to handle specific operations in lattice-based encryption schemes more efficiently. For instance, [BDH+21] and [CGMZ23] proposed specialized gadgets for masking the Kyber lattice-based encryption scheme when performing ciphertext comparison. In general, the use of lattice-based cryptography has led to the development of new dedicated gadgets to perform high-order masked computation more efficiently.

Masking Dilithium signatures. The first high-order masking of a lattice-based signature scheme was presented in [BBE+18], which focused on masking the GLP signature scheme [GLP12]. Dilithium is a more advanced version of GLP, using the compression technique from [BG14] and additional optimizations. In [MGTF19], the authors demonstrated how to perform high-order masking of Dilithium. Furthermore, the authors presented a simplified

version of Dilithium that used a power-of-two modulus q instead of a prime modulus, resulting in significant performance improvements for their countermeasure.

In a signature generation algorithm, not all variables necessarily need to be masked. For Dilithium, the authors of [ABC+22] revisited the sensitivity analysis presented in [MGTF19] and identified some intermediate variables that were incorrectly left unmasked in [MGTF19], which could potentially lead to the recovery of the private key. Conversely, they also showed that some other variables in [MGTF19] were unnecessarily protected. More specifically, the authors of [ABC+22] argue that the vector $\mathbf{w} = \mathbf{Ay}$ must be masked, as opposed to what was done in [MGTF19]. If left unmasked, since $\mathbf{A}$ is either a square matrix or has more rows than columns, the variable $\mathbf{y}$ can be efficiently recovered from $\mathbf{w}$, which in turn enables the recovery of the secret key. On the other hand, the variable $\mathbf{r} = \mathbf{w} - c\mathbf{s}_2 = \mathbf{Az} - c\mathbf{t}$ can be considered public after the rejection sampling. Therefore, the computation of the hint vector $\mathbf{h}$ can be performed in the clear and need not be masked, as was done in [MGTF19]. In this paper, we follow the same approach as [ABC+22].

The authors of [ABC+22] have also proposed improved gadgets for the high-order masking of Dilithium. Specifically, they have presented an efficient high-order algorithm for performing the rejection sampling of the variables $\mathbf{z}$ and $\tilde{\mathbf{r}}$, as well as an efficient masking technique for the Decompose function. They have argued that the variable $\mathbf{w}_1$ in $(\mathbf{w}_1, \mathbf{w}_0) \leftarrow \mathsf{Decompose}_q(\mathbf{w})$ need not be masked since it is also publicly computed during signature verification. Thus, the challenge $\tilde{c} = H(M \| \mathbf{w}_1)$ need not be masked, and the Keccak hash function H need not be masked. However, this may not hold for an aborted signature, where knowledge of $\mathbf{w}_1$ might reveal information to an attacker. The heuristic assumption (also used in [BBE+18] and [MGTF19]) is that revealing the commitment $\mathbf{w}_1$ of the aborted transcript does not compromise the security of the scheme. Recently, this assumption has been analyzed in [DFPS23] and shown to hold unconditionally by providing an efficient simulator for all transcripts, including aborted ones. Therefore, in this work, we adopt the same approach.

Finally, the authors considered the masking of both deterministic Dilithium and randomized Dilithium. In this paper, for simplicity we only consider the masking of randomized Dilithium, since as demonstrated in [ABC+22] its masking is significantly more efficient.

Our contributions. We describe improved high-order gadgets for the masking of Dilithium, with a proof of security in the t-probing model. Additionally, we provide an implementation to demonstrate the effectiveness of our countermeasures. More specifically:

- We introduce a new gadget called ShiftMod in Section 3.1 that allows for efficient arithmetic shifts. Compared to previous works such as [CGMZ22], our ShiftMod algorithm is more versatile since it can handle any integer modulus $2q$ instead of just 2^k. Our new gadget is particularly useful in the context of Dilithium, where we utilize it to develop a fast Boolean-to-arithmetic modulo q conversion, as well as a faster masking of the Decompose function (discussed below). Additionally, we can replace the arithmetic shift gadgets (Shift1 and Shift2) from [CGMZ22] with our ShiftMod algorithm, which is more efficient, to expedite the conversion from arithmetic modulo 2^k to Boolean masking.

- We propose a new algorithm to convert a μ-bit integer x from Boolean to arithmetic masking modulo any integer q, which is used in Dilithium for generating an arithmetic sharing of the random variable y modulo q. Unlike existing state-of-the-art methods, our algorithm exhibits a complexity that is independent of both the bit-size μ and the modulus q, assuming an arithmetic operation modulo q has a unit cost. Our approach builds upon the work of [BCZ18], which was limited to power-of-two moduli, and extends it to handle arbitrary moduli q. As in [BCZ18], the complexity of our

algorithm is $\mathcal{O}(2^n)$, which is exponential in the number of shares n but independent of the modulus size. For small values of n and large enough μ, our method is also significantly faster than alternative techniques such as [BBE+18] and [SPOG19].

Our technique first applies [BCZ18] for a well-chosen modulus 2^k, and then performs a modulus switching of the arithmetic shares from modulo 2^k to modulo q. However, such modulus switching can introduce a small error e. To address this, we utilize our previous ShiftMod algorithm to remove the error e and obtain an exact conversion.

- For masking the Decompose algorithm of Dilithium, we improve the two existing approaches introduced in [ABC+22]. Firstly, we expand on the first approach based on the Compress function of Kyber, by providing an alternative description of the Decompose procedure used in Dilithium, and a more efficient masking based on our ShiftMod algorithm. We also extend the second approach to cover any $\alpha = (q-1)/\delta$, whereas in [ABC+22], a power-of-two δ was required. Furthermore, we compare the two approaches and show that the first approach is more efficient in practice.

Finally, we present a complete high-order masked implementation of Dilithium, utilizing the improved gadgets described above. We provide the practical results of a C implementation and compare the performance improvement provided by our new gadgets with those from [BBE+18], [SPOG19], and [ABC+22]. For a non-bitsliced implementation, our techniques achieve a significant speedup compared to previous work. In particular, our new Boolean-to-arithmetic masking algorithm leads to a substantial improvement over the techniques presented in [SPOG19] and [BBE+18]. Our improved approaches for masking the Decompose algorithm of Dilithium also provides better efficiency than the ones presented in [ABC+22]. The plain C code is publicly available at

https://github.com/fragerar/Masked_Dilithium

Comparison with [ABC+22]. In the following, we provide a more detailed comparison of our contributions with those of [ABC+22]. Note that [ABC+22] also describes a leveled approach combining the shuffling countermeasure with masking, which offers significantly better performances, but without the guarantee of t-probing security. Here we only consider the fully masked implementation from [ABC+22]. As previously mentioned, we use the same global masking strategy as in [ABC+22], which involves masking the same variables and keeping the same variables unmasked.

Regarding the generation of the signing nonce y, the authors of [ABC+22] rely on the Boolean to arithmetic conversion from [BBE+18], with the bitsliced implementation from [BC22], while we use our new Boolean to arithmetic conversion algorithm from Section 4. Although our algorithm has a complexity independent from the bit-size of y and q, it is not compatible with bitslicing.

Similarly, for the masking of Decompose, we describe improved gadgets in Section 5, which can leverage our more lightweight ShiftMod arithmetic shift algorithm. Our ShiftMod gadget is more efficient compared to the heavier arithmetic to Boolean conversion from [BBE+18] used in [ABC+22], but it is not compatible with bitslicing.

Moreover, the authors of [ABC+22] provide a complete benchmark for an ARM Cortex-M4 microcontroller for the full signature generation, while we only provide a C implementation for laptop execution. However, we provide a public implementation, whereas the source code for [ABC+22] is not available.

In summary, the main difference with [ABC+22] is that we provide more efficient gadgets for a non-bitsliced implementation, whereas [ABC+22] use more standard gadgets for which they can leverage the state-of-the-art bitsliced implementation from [BC22]. We provide in Section 7.1 a high-level comparison with the bitsliced approach of [BC22] for the high-order Boolean to arithmetic conversion modulo q.

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 will use both the positive representation of elements in $\mathbb{Z}_q$ (i.e., $\mathbb{Z}_q \simeq \{0, \ldots, q-1\}$) and the centered representation (i.e., $\mathbb{Z}_q \simeq \{-(q-1)/2, \ldots, 0, \ldots, (q-1)/2\}$ for odd q and $\mathbb{Z}_q \simeq \{-(q/2)+1, \ldots, 0, \ldots, q/2\}$ for even q). 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 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:

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

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 $\|\mathbf{z}\|_\infty = \max_{1 \le i \le k} \|z_i\|_\infty$.

The centered ball of radius η is denoted by S_η and consists of elements $w \in R$ such that $\|w\|_\infty \le \eta$, or equivalently, polynomials of R with coefficients in the range $[-\eta, \eta]$. Finally, we denote by $\tilde{S}_\eta$ the polynomials of R with coefficients in the range $]-\eta, \eta]$.

2.2 Security definitions

In the following, we review the Non-Interference (NI) and Strong Non-Interference (SNI) security notions proposed in [BBD+16], along with the stronger free-SNI notion introduced in [CS21]. These security notions are particularly useful for demonstrating probing security against attackers who can only probe a limited number of variables. In fact, by using composition theorems established in [BBD+16], we can prove that an algorithm achieves probing security by breaking it down into smaller gadgets for which we individually prove either NI or SNI security.

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 $|I| \le t_1$.

The NI and SNI notions differ in the number of input shares required for the simulation. In the SNI case, this number is upper-bounded by the number of internal probes in the gadget. In contrast, for the NI notion, it is only bounded by the total number of probes, including the probes on the output shares. Finally, we also review the free-SNI notion introduced in [CS21]. This notion is stronger than the SNI notion, as a subset of the output shares can be perfectly simulated from a subset of the input shares, and all other output shares, except one, can be simulated using fresh uniform randoms.

Definition 3 (Free-t-SNI security)**.** Let G be a gadget taking as input n shares $(a_i)_{1 \le i \le n}$ and outputting n shares $(b_i)_{1 \le i \le n}$. The gadget G is said to be free t-SNI secure if for any set of $t_1 \le t$ probed intermediate variables, there exists a subset I of input indices

with $|I| \leq t_1$, such that the t_1 intermediate variables and the output variables $b_{|I}$ can be perfectly simulated from $a_{|I}$, while for any $O \subsetneq [1, n] \setminus I$ the output variables in $b_{|O}$ are uniformly and independently distributed, conditioned on the probed variables and $b_{|I}$.

We recall in Appendix A.2 the mask refreshing algorithm RefreshMasks. The algorithm was proven $(n-1)$-SNI in [BBD+16]. The lemma below shows that it also satisfies the stronger free-SNI property.

Lemma 1 ([CS21]). *The* RefreshMasks *algorithm is free-*$(n-1)$*-SNI.*

For masking Dilithium, the shares must be eventually recombined to output the signature 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 4 (t-NIo security [BBE+18]). Let G be a gadget taking as input $(x_i)_{1 \leq i \leq n}$ and outputting b. The gadget G is said t-NIo secure if for any set of $t_1 \leq t$ intermediate variables, there exists a subset I of input indices with $|I| \leq t_1$, such that the t_1 intermediate variables can be perfectly simulated from $x_{|I}$ and b.

To satisfy this definition when the output shares of a gadget need to be recombined with a public output, as shown in [CGMZ23, Appendix B.2], one can apply the free-SNI Refreshmasks algorithm and eventually recombine the shares.

3 Arithmetic shift and conversion from arithmetic to Boolean masking

In this section, we introduce a novel gadget called ShiftMod that enables efficient arithmetic shifts modulo any integer. The purpose of this gadget is to generate an arithmetic sharing of $\lfloor x/2 \rfloor$ modulo q, given an arithmetic sharing of x modulo $2q$. We provide a detailed description of the properties of ShiftMod in Section 3.1.

Compared to the arithmetic shift gadgets introduced in [CGMZ22], our ShiftMod algorithm is more versatile as it can operate modulo any integer $2q$, not just 2^k. This enables the development of other gadgets for the masking of Dilithium, such as a fast Boolean-to-arithmetic modulo q conversion algorithm, which is detailed in Section 4. This conversion algorithm is crucial in generating an arithmetic sharing of the random polynomial y modulo q in Dilithium. Moreover, we also utilize ShiftMod to improve the masking of the Decompose function in Dilithium, as described in Section 5.

Additionally, we demonstrate that our ShiftMod algorithm can be applied to expedite the conversion from arithmetic modulo 2^k to Boolean masking, even though this conversion is not directly used in our masking of Dilithium. For this, we utilize the same technique presented in [CGMZ22]. We apply a sequence of arithmetic shifts starting from x modulo 2^k, and extract the least significant bit of $\lfloor x/2^i \rfloor$ at each step. This ultimately yields a Boolean masking of x. By employing our new ShiftMod gadget, we achieve significant improvements in conversion efficiency compared to the two arithmetic shift gadgets (Shift1 and Shift2) proposed in [CGMZ22].

3.1 New arithmetic shift modulo $2q$

We consider an arbitrary integer q. Given as input an arithmetic sharing of $x = x_1 + \cdots + x_n \pmod{2q}$, our new ShiftMod gadget high-order computes the shift:

$$a = \left\lfloor \frac{x}{2} \right\rfloor = \frac{x - (x \bmod 2)}{2} \pmod{q} \tag{1}$$

and returns an arithmetic sharing of $a = a_1 + \cdots + a_n \pmod q$, without leaking information about x.

Our ShiftMod algorithm relies on a 1-bit Boolean-to-arithmetic conversion gadget, with the most efficient one being the 1bitB2A gadget from [SPOG19], which is proven SNI. However, to use this gadget in our ShiftMod algorithm, we require a slightly stronger property than SNI, namely free-SNI (see Definition 3). One approach to achieving free-SNI is to compose the 1bitB2A gadget from [SPOG19] with the RefreshMasks algorithm, as it achieves this property (see Lemma 1). However, we demonstrate that a minor modification to the 1bitB2A gadget directly achieves the free-SNI property, eliminating the need for RefreshMasks. This modification results in a more efficient ShiftMod algorithm.

To high-order compute (1), the first step is to obtain an arithmetic masking modulo $2q$ of $y = (x \bmod 2)$. For this, we consider the least significant bits b_i of x_i for $1 \le i \le n$, and we perform a Boolean to arithmetic modulo $2q$ conversion of the shares b_i, using the 1bitB2A algorithm from [SPOG19], which we recall in Section 3.2 (see Alg. 2). This gives:

$$y_1 + \cdots + y_n = b_1 \oplus \cdots \oplus b_n = (x \bmod 2) \pmod{2q}$$

In the second step, we high-order compute $z = x - (x \bmod 2) \pmod{2q}$ by letting $z_i = x_i - y_i \bmod 2q$ for all $1 \le i \le n$. To high-order compute $a = z/2 \bmod q$, our technique is to obtain a new arithmetic sharing of z such that $z_i = 0 \pmod 2$ for all $1 \le i \le n$; then we can simply divide by 2 each share z_i independently. For this, we do a loop for $i = 1$ to $n - 1$, and we let $z_n \leftarrow z_n + (z_i \bmod 2) \bmod 2q$, and $z_i \leftarrow z_i - (z_i \bmod 2) \bmod 2q$. At the end of the loop, the shares z_i still encode the same integer z, and all the shares z_i are even. This is true by construction of the updated z_i's for $1 \le i \le n-1$, and this must also be true for z_n because $z = 0 \pmod 2$. Therefore, we can eventually let $a_i = z_i/2$ for all $1 \le i \le n$, which gives an arithmetic sharing modulo q of $a = \lfloor x/2 \rfloor \pmod q$. We obtain the following ShiftMod algorithm depicted in Algorithm 1.

Note that the ShiftMod algorithm we propose can handle any integer q, including those for which $\gcd(q, 2) \neq 1$. As a result, computing $a = z/2 \bmod q$ using high-order methods is not as simple as multiplying the shares z_i of z by $2^{-1} \bmod q$. In particular, to perform an arithmetic shift by k bits, we will use the ShiftMod gadget iteratively with moduli of the form $q = 2^i \cdot p$ for i decreasing from k to 1. Therefore, we cannot assume $\gcd(q, 2) = 1$.

Algorithm 1 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: $(y_1, \ldots, y_n) \leftarrow$ 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: $\quad z_n \leftarrow z_n + (z_i \,\&\, 1) \bmod 2q$
6: $\quad z_i \leftarrow z_i - (z_i \,\&\, 1) \bmod 2q$
7: **end for**
8: **for** $i = 1$ to n **do** $a_i \leftarrow z_i \gg 1$
9: **return** $a_1, \ldots, a_n$

Complexity. We recall the 1bitB2A algorithm in Section 3.2, and show that its number of elementary operations is $T_{\mathsf{1bitB2A}}(n) = 2n^2 + 4n - 6$. The number of operations of ShiftMod is then $T_{\mathsf{ShiftMod}}(n) = n + T_{\mathsf{1bitB2A}}(n) + n + 3(n-1) + n = 2n^2 + 10n - 9$, assuming that operations modulo q have unit cost. The complexity is therefore $2n^2 + \mathcal{O}(n)$, which is significantly better than the previous Shift1 and Shift2 algorithms from [CGMZ22], which

have complexity $2n^3$ and $20n^2$ respectively (neglecting low-order terms). Therefore, the resulting arithmetic to Boolean conversion algorithm will be significantly more efficient (see Section 3.3). Moreover, our ShiftMod gadget is more general, as it can work modulo any integer $2q$, instead of modulo 2^k only for Shift1 and Shift2.

Security. To prove the t-NI property of the ShiftMod algorithm above, it is sufficient to prove that after Line 3, we can perfectly simulate all variables $(z_i \,\&\, 1)$ for $1 \leq i \leq n$, since the other operations (except 1bitB2A) compute the shares independently for each i. For this, it is sufficient to show that after Line 3, all output shares z_i except one can be perfectly simulated; then, using the relation $\sum_{i=1}^{n} z_i = 0 \pmod 2$, we can perfectly simulate all variables $(z_i \,\&\, 1)$ for $1 \leq i \leq n$ as required. For this, we crucially use the free-SNI property of 1bitB2A, which implies that for any subset $O \subsetneq [1, n] \setminus I$, the variables $y_{|O}$ are uniformly and independently distributed. Then such variables y_i can play the role of a one-time-pad at Line 3 with respect to the variables x_i for $i \notin I$, and therefore all variables z_i except one can be perfectly simulated, as required.

As required, we prove the free-SNI property of the 1bitB2A gadget from [SPOG19] in Section 3.2. As explained previously, without the free-SNI property of 1bitB2A, a full mask refreshing of the shares z_i would have been required after Line 3, to get the free-SNI property of the composition (see Lemma 1). This would have added a term $3n^2/2$ in the number of operations of ShiftMod, which would then be $7n^2/2$ instead of $2n^2$ (neglecting low order terms).

Theorem 1. *The* ShiftMod *algorithm achieves the* NI *property, if the* 1bitB2A *algorithm achieves the free-*SNI *property.*

Proof. We provide a proof based on simulation, namely, given any set of t probes, one constructs iteratively a subset I of indices of the input shares x_i that are sufficient to simulate the t probes. Then by ensuring $|I| \leq t$, we will deduce that the simulation can be performed without knowing the original variable x when $t < n$. We let t_1 be the variables probed everywhere in the algorithm apart from 1bitB2A and t_2 the probed variables in 1bitB2A, such that $t = t_1 + t_2$. In the following, we assume that $t < n$, otherwise we can trivially simulate all probes with $I = [1, n]$, which amounts to knowing x.

We describe hereafter the construction of the set $I \subset [1, n]$ initially empty. More precisely, we let $I = J \cup U$ and we describe the construction of the sets J and U. For every probed variable x_i, b_i, y_i, z_i or a_i, we add i to J. For every probed variable $z_{n,i}$ corresponding to the result variable z_n at loop i, we add n to J. Furthermore, the set U is constructed from the free-SNI property of 1bitB2A (Theorem 2) which allows to simulate the t_2 probed intermediate variables in 1bitB2A and the output variables $y_{|U}$ from a set U of indices where $|U| \leq t_2$. By construction, since one has added at most one index per probed variable in J, we have $|J| \leq t_1$. This gives $|I| = |J \cup U| \leq t_1 + t_2 = t$ as required.

The simulation of the probed variables is done as follows: if x_i or b_i is probed, then it can perfectly be simulated from the knowledge of x_i since $i \in J \subset I$ by construction. Then, for $i \in U$, the output variables y_i are simulated from the free-SNI property of 1bitB2A. In the case where $i \notin U$, because we have $t < n$, we know that there is at least one index $i^\star$ which does not belong to I. Therefore, the free-SNI property of 1bitB2A allows us to construct a set of output indices $O = [1, n] \setminus (U \cup \{i^\star\}) \subsetneq [1, n] \setminus U$ such that the variables $y_{|O}$ are uniformly and independently distributed. Therefore, the variables y_i such that $i \in J \setminus U$ can be simulated by generating random values. The remaining variables y_i, namely with $i \notin I$, can play the role of a one-time-pad at line 3 of ShiftMod with respect to the variables x_i for $i \notin I$, and therefore all variables z_i except $z_{i^\star}$ at Line 3 can be perfectly simulated.

The simulation continues as follows: any probed variable z_i or $z_i \,\&\, 1$ with $1 \leq i \leq n-1$ in the For loop, can be perfectly simulated as shown right above, since one has at most $n - 1$ such probes.

It remains to perfectly simulate any probed intermediate variable $z_{n,i}$ for any i when $n \in I$. To this aim, we use the fact that the only values which enter into the computation of $z_{n,i}$ are the first z_n value (namely $z_{n,0}$), and the least significant bit of all z_i's (namely $z_i \,\&\, 1$ for $1 \leq i \leq n$). Since we can perfectly simulate $n-1$ variables z_i and because we know that the sum of all z_i's is even, we can deduce the least significant bit of the unknown value $z_{i^\star}$, namely $z_{i^\star} \,\&\, 1 = \bigoplus_{i \neq i^\star} (z_i \,\&\, 1)$. Thus, we have shown that we can perfectly simulate any probed intermediate variable $z_{n,i}$, including the output result z_n at the end of the For loop.

Eventually, the probed variable a_i can also be directly computed from z_i above since $i \in I$ by construction, which concludes the proof. □

3.2 1-bit Boolean to arithmetic conversion from [SPOG19]

In this section, we recall the 1-bit Boolean to arithmetic conversion algorithm 1bitB2A from [SPOG19], which is used in our ShiftMod algorithm above. This algorithm is currently the most efficient of its kind and has the same asymptotic complexity $\mathcal{O}(n^2)$ for n shares as the table-based algorithm from [CGMZ22], but with a better concrete complexity. We will demonstrate that with some minor modifications, the [SPOG19] algorithm can achieve the free-SNI property required for our ShiftMod gadget.

For any integer q, the algorithm takes as input n shares $x_i \in \{0,1\}$ and outputs n arithmetic shares $y_i \in \mathbb{Z}_q$ such that $y_1 + \cdots + y_n = x_1 \oplus \cdots \oplus x_n \pmod q$. It works as follows. Assume that we have a n-shared encoding $(y_1, \ldots, y_n)$ modulo q of a bit $b \in \{0,1\}$, that is $b = y_1 + \cdots + y_n \pmod q$. Then, we can easily compute an encoding of $\bar{b} = 1 - b$, simply by letting $y_1 \leftarrow 1 - y_1 \bmod q$ and $y_i \leftarrow -y_i \bmod q$ for all $2 \leq i \leq n$. Using this technique, starting from an encoding $(y_1, \ldots, y_n)$ of 0, we can iteratively process the input bits x_i, and eventually obtain an encoding $(y_1, \ldots, y_n)$ such that $y_1 + \cdots + y_n = x_1 \oplus \cdots \oplus x_n \pmod q$ as required. For security, we must refresh the shares y_j after the processing of each x_i. One uses a mask refreshing LinearRefresh that accumulates the randomness on the last share; we recall its description in Appendix A.1. As an optimization, instead of starting with n shares y_j, it is actually more efficient to progressively increase the number of shares, from a single share to n shares.

We recall the corresponding algorithm from [SPOG19] in Alg. 2 below. To facilitate the writing of our security proof of the free-SNI property, we include a minor modification. Namely, we start the processing of the input Boolean shares with x_2, instead of x_1; we then process the remaining shares $x_3, \ldots, x_n$, and eventually $x_{n+1} = x_1$. Moreover, the last LinearRefresh is performed with reversed inputs, so that the randomness is accumulated on v_1 instead of v_n. This cyclic shift of the inputs is to ensure that after the processing of x_i, the randomness in LinearRefresh is always accumulated on the i-th column.

Algorithm 2 1bitB2A [SPOG19]

Input: $x_1, \ldots, x_n \in \{0,1\}$
Output: $v_1, \ldots, v_n \in \mathbb{Z}_q$ such that $v_1 + \cdots + v_n \bmod q = x_1 \oplus \cdots \oplus x_n$

1: $x_{n+1} \leftarrow x_1$, $v_1 \leftarrow x_2$ ▷ $v_1 = x_2 \pmod q$
2: **for** $i = 2$ to n **do**
3: $\quad (v_1, \ldots, v_i) \leftarrow \mathsf{LinearRefresh}_{\mathbb{Z}_q}(v_1, \ldots, v_{i-1}, 0)$
4: $\quad (v_1, \ldots, v_i) \leftarrow (1 - 2x_{i+1}) \cdot (v_1, \ldots, v_i) \bmod q$
5: $\quad v_1 \leftarrow v_1 + x_{i+1} \bmod q$ ▷ $\sum_{j=1}^{i} v_i = x_2 \oplus \cdots \oplus x_{i+1} \pmod q$
6: **end for**
7: $(v_n, \ldots, v_1) \leftarrow \mathsf{LinearRefresh}_{\mathbb{Z}_q}(v_n, \ldots, v_1)$
8: **return** $(v_1, \ldots, v_n)$

Complexity. The number of operations of LinearRefresh is $3i-3$ for i shares. Therefore, the total number of operations is $T_{\mathsf{1bitB2A}}(n) = \sum_{i=2}^{n}(3i-3+i+2)+3n-3 = 2n^2+4n-6$. More generally, to convert k-bit of Boolean masking into arithmetic masking, the complexity is $T_{\mathsf{B2A}}(k,n) = k \cdot (4n + T_{\mathsf{1bitB2A}}(n)) = k \cdot (2n^2+8n-6)$.

Security. Algorithm 2 is already proven SNI in [SPOG19]. With Theorem 2, we prove a slightly stronger property, namely the free-SNI property (see Def. 3); we provide the proof in Appendix B. As explained previously, this enables to obtain a better asymptotic complexity for our ShiftMod algorithm.

Theorem 2 (free-SNI of 1bitB2A)**.** *For any set of t probes, there exists a subset I with $|I| \leq t$, such that those t probes and $v_{|I}$ can be perfectly simulated. Moreover the shares in $v_{|O}$ are uniformly and independently distributed for any $O \subsetneq [1,n] \setminus I$, even conditioned on the probes and $v_{|I}$.*

3.3 Improved arithmetic to Boolean conversion

Finally, we show how to use our ShiftMod gadget for faster conversion from arithmetic to Boolean masking. Namely, in [CGMZ22], the authors described the arithmetic to Boolean conversion algorithm below, based on a generic high-order 1-bit arithmetic Shift, working modulo 2^k. Therefore, in the algorithm below, we can replace the Shift1 or Shift2 algorithms from [CGMZ22] by our improved ShiftMod gadget form Section 3.1. In that case, the ShiftMod algorithm takes as input a modulus $q' = 2^{k-j} = 2q$.

Algorithm 3 ArithmeticToBoolean (ABoptiNI) [CGMZ22]

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

1: **for** $i = 1$ to n **do** $s_i \leftarrow 0$
2: **for** $j = 0$ to $k-1$ **do**
3: **for** $i = 1$ to n **do** $s_i \leftarrow s_i + ((z_i \,\&\, 1) \ll j)$
4: $(z_1, \ldots, z_n) \leftarrow \mathsf{ShiftMod}(2^{k-j}, (z_1, \ldots, z_n))$
5: **end for**
6: **return** $s_1, \ldots, s_n$

Complexity. The complexity is $T_{\mathsf{AB}}(n,k) = k \cdot (3n + T_{\mathsf{shift}}(n))$. Therefore, with the ShiftMod algorithm, the complexity is $T_{\mathsf{AB}}(n,k) = k \cdot (2n^2+13n-9)$. The complexity is therefore $2kn^2 + \mathcal{O}(kn)$, instead of $2kn^3$ and $20kn^2$ with Shift1 and Shift2 respectively (neglecting low-order terms). We provide a comparison in Table 1 below. We see that our new algorithm outperforms [CGV14] for security order $t \geq 4$.

Theorem 3 ($t-$NI security of ABoptiNI)**.** *For any set of t intermediate variables, there exists a subset of input indices $I \subset [1,n]$ such that the t intermediate variables can be perfectly simulated from inputs $z_{|I}$, with $|I| \leq t$.*

Proof. The proof is straightforward as Algorithm 3 is the composition of the $t-$NI ShiftMod algorithm with sharewise operations. □

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

A mod 2^{32} → B	Security order t								
	1	2	3	4	5	6	8	10	12
Goubin [Gou01]	165								
[CGV14] 32 → 32		1 132	2 070	4 030	6 218	8 597	15 053	23 655	33 572
Alg. 3 with Shift1		2 496	5 248	9 600	15 936	24 640	50 688	90 816	148 096
Alg. 3 with Shift2		3 872	7 520	12 448	18 656	26 144	44 960	68 896	97 952
Alg. 3 with ShiftMod		1 536	2 400	3 392	4 512	5 760	8 640	12 032	15 936

4 Boolean to arithmetic conversion modulo q with size-independent complexity

In this section, we introduce a new algorithm that converts a μ-bit integer x from Boolean to arithmetic masking modulo any integer q. This conversion is used in Dilithium to generate an arithmetic sharing of the variable y modulo q. Currently, the state of the art method involves applying the 1-bit Boolean to arithmetic conversion algorithm from [SPOG19] μ times, resulting in a complexity of $\mathcal{O}(n^2 \cdot \mu)$. Alternatively, one can use the Boolean to arithmetic conversion algorithm from [BBE+18], which has a complexity of $\mathcal{O}(n^2 \cdot \log q)$, or even $\mathcal{O}(n^2 \cdot \log \log q)$ using the Kogge-Stone adder as in [CGTV15].

Our contribution is to describe a Boolean to arithmetic conversion algorithms with complexity independent of both the bitsize μ and the modulus q. We assume that an arithmetic operation modulo q has a unit cost. We build on the work of [BCZ18], which described a μ-bit Boolean to arithmetic modulo 2^k conversion algorithm with complexity $\mathcal{O}(2^n)$, exponential in the number of shares n, but independent of the bitsize μ and the register size k. For small values of n, this method is at least one order of magnitude faster than alternative techniques. We generalize this algorithm to any modulus q, and as in [BCZ18], we obtain a significant improvement for small n values, for large enough μ as in Dilithium.

We start with an approximate algorithm that can induce a small error e in the conversion, based on modulus switching. For this, we first apply the conversion of [BCZ18] from a μ-bit Boolean masking into an arithmetic modulo 2^k masking, for a certain parameter $k \geq \mu$. In a second step, we convert from arithmetic masking modulo 2^k to modulo q, using modulus switching, with complexity $\mathcal{O}(n)$. To obtain an exact algorithm, we show how to remove the small error e from the approximate algorithm, using our ShiftMod gadget from Section 3.1, with additional complexity $\mathcal{O}(n^2 \log n)$. The total complexity remains $\mathcal{O}(2^n)$ and is independent of the modulus size as in [BCZ18]. In summary, our new algorithm works for any modulus q, whereas [BCZ18] only worked for a power-of-two modulus.

In the context of Dilithium, we must generate an arithmetically masked polynomial y modulo q, with randomly distributed coefficients in $]-\gamma_1, \gamma_1]$, with $\gamma_1 = 2^{17}$ for security level 2. Therefore, for each coefficient, we first generate a random μ-bit Boolean masked x, simply by generating n uniformly random μ-bit Boolean shares. By applying our exact conversion algorithm, we obtain an arithmetically masked x modulo q, with uniform distribution in $[0, 2^\mu[$. Therefore, we set $\mu = 18$, and we subtract $\gamma_1 - 1 = 2^{17} - 1$ from the first share to obtain the required uniform distribution in $]-\gamma_1, \gamma_1]$.

4.1 Tool: modulus switching

We begin by introducing our primary method: modulus switching over arithmetic shares. This technique was already used in [CGMZ23] to mask the Compress function of Kyber.

In this section, we provide an analysis of the fundamental modulus switching approach. Specifically, we examine the error that arises when n arithmetic shares are switched from a modulus p_1 to a modulus p_2. We demonstrate that the error e is confined within the range of $0 \leq e \leq n-1$.

Algorithm 4 Modulus switching (ModSwitch)

Input: $p_1, p_2 \in \mathbb{N}$, an arithmetic masking $x_1, \ldots, x_n \in \mathbb{Z}_{p_1}$ such that $x = x_1 + \cdots + x_n \pmod{p_1}$.

Output: $y_1, \ldots, y_n \in \mathbb{Z}_{p_2}$ such that $y_1 + \cdots + y_n = \lfloor x \cdot p_2/p_1 \rfloor + e \bmod p_2$, for $0 \leq e < n$.

1: **for** $i = 1$ **to** n **do** $y_i \leftarrow \lfloor x_i \cdot p_2/p_1 \rfloor \bmod p_2$
2: $y_1 \leftarrow y_1 + (n-1) \bmod p_2$
3: **return** $(y_1, \ldots, y_n)$

Lemma 2. *Let p_1 and p_2 be two positive integers. Let $x_i \in \mathbb{Z}_{p_1}$ for $1 \leq i \leq n$ and let $x = x_1 + \cdots + x_n \pmod{p_1}$. Let $y_1, \ldots, y_n$ be the output of* ModSwitch. *Then:*

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

for some $0 \leq e \leq n-1$.

Proof. We write $x = x_1 + \cdots + x_n - \lambda \cdot p_1$ for some $\lambda \in \mathbb{Z}$, so we can write:

$$\frac{x \cdot p_2}{p_1} = \sum_{i=1}^{n} \frac{x_i \cdot p_2}{p_1} - \lambda \cdot p_2$$

For $1 \leq i \leq n$, we can write by Euclidean division $x_i \cdot p_2 = p_1 \cdot \lfloor x_i \cdot p_2/p_1 \rfloor + r_i$ with $0 \leq r_i < p_1$, and similarly $x \cdot p_2 = p_1 \cdot \lfloor x \cdot p_2/p_1 \rfloor + r$ with $0 \leq r < p_1$. This gives:

$$\left\lfloor \frac{x \cdot p_2}{p_1} \right\rfloor = \sum_{i=1}^{n} \left\lfloor \frac{x_i \cdot p_2}{p_1} \right\rfloor + \sum_{i=1}^{n} \frac{r_i}{p_1} - \frac{r}{p_1} - \lambda \cdot p_2$$

Moreover, since $x = \sum_{i=1}^{n} x_i \pmod{p_1}$, we must have $r = \sum_{i=1}^{n} r_i \pmod{p_1}$, and therefore $r = \sum_{i=1}^{n} r_i - e' \cdot p_1$ for some $e' \in \mathbb{Z}$, with $0 \leq e' \leq n-1$. This gives:

$$\begin{aligned} \left\lfloor \frac{x \cdot p_2}{p_1} \right\rfloor &= \sum_{i=1}^{n} \left\lfloor \frac{x_i \cdot p_2}{p_1} \right\rfloor + e' - \lambda \cdot p_2 \\ &= y_1 + \cdots + y_n - (n-1) + e' - \lambda \cdot p_2 \end{aligned}$$

Finally, by letting $e = (n-1) - e'$ and after reduction modulo p_2, we obtain (2). □

4.2 Approximate conversion using modulus switching

In the following we describe an approximate conversion algorithm based on modulus switching that can induce a small error e in the conversion. More precisely, we are given as input a Boolean masking of x, such that $0 \leq x < 2^{\mu}$, with $x = u_1 \oplus \cdots \oplus u_n$. Our goal is to obtain an arithmetic masking of $x + e$ modulo q, for a small error $e \in \mathbb{Z}$ with $0 \leq e \leq n-1$:

$$y_1 + \cdots + y_n = x + e \pmod{q}$$

We assume that μ and the modulus q are fixed (for example, $\mu = 18$ and $q = 2^{23} - 2^{13} + 1$ in Dilithium). Note that we can get a centered error by letting $y_1' := y_1 - \lfloor n/2 \rfloor$, which gives $|e'| \leq \lfloor n/2 \rfloor$.

We first apply the [BCZ18] Boolean to arithmetic conversion algorithm, which gives $x = x_1 + \cdots + x_n \pmod{2^k}$, for a parameter k that will be determined later. We then apply Lemma 2 by performing a modulus switching from the modulus $p_1 = 2^k$ to a new modulus $p_2 = a \cdot q$, for some integer a that will also be determined later, and we reduce the shares y_i directly modulo q. More precisely, given as input the shares x_i modulo 2^k such that $x = x_1 + \cdots + x_n \pmod{2^k}$, we compute for $1 \leq i \leq n$:

$$y_i = \left\lfloor \frac{x_i \cdot a \cdot q}{2^k} \right\rceil \bmod q$$

and $y_1 \leftarrow y_1 + (n-1)$, as in Lemma 2. After reduction modulo q, from (2) we obtain for some $0 \leq e \leq n-1$:

$$y_1 + \cdots + y_n = \left\lfloor \frac{x \cdot a \cdot q}{2^k} \right\rceil + e \pmod{q} \tag{3}$$

Equation (3) is not sufficient for our purposes as, due to the modulus switching, it only gives us an arithmetic sharing modulo q of $\lfloor x \cdot a \cdot q / 2^k \rceil$ instead of x. To address this issue, we use a trick: we select an integer a such that $a \cdot q$ is very close to 2^k, which allows us to obtain the correct value of x.

More precisely, our goal is now to ensure that for any $0 \leq x < 2^\mu$, we have

$$\left\lfloor \frac{x \cdot a \cdot q}{2^k} \right\rceil = x \tag{4}$$

which using (3) will give $y_1 + \cdots + y_n = x + e \pmod{q}$. This is as required an arithmetic sharing of x modulo q, up to some small error e. The following lemma ensures that when the integer $a \cdot q$ is sufficiently close to 2^k, Equality (4) holds as the modulus switching does not change the value of x.

Lemma 3. *Let $k := \lceil \log_2 q \rceil + \mu$ and $a := \lceil 2^k / q \rceil$. Then for any $0 \leq x < 2^\mu$, we have $\lfloor x \cdot a \cdot q / 2^k \rceil = x$.*

Proof. We must ensure that for any $0 \leq x < 2^\mu$, we have $0 \leq x \cdot a \cdot q / 2^k - x < 1$. It suffices to ensure that this holds for the upper-bound $x = 2^\mu$, that is $0 \leq 2^\mu \cdot a \cdot q / 2^k - 2^\mu < 1$, which gives the sufficient condition $0 \leq a \cdot q - 2^k < 2^{k-\mu}$. From $a := \lceil 2^k / q \rceil$, we get $2^k = q \cdot a - r$ for $0 \leq r < q$, which gives $0 \leq q \cdot a - 2^k < q$. From $k := \lceil \log_2 q \rceil + \mu$, we get $2^k \geq q \cdot 2^\mu$, which gives $0 \leq q \cdot a - 2^k < q \leq 2^{k-\mu}$ as required. □

This gives the following $\mathsf{BtoA_qApprox}$ algorithm depicted in Algorithm 5. We denote by $\mathsf{BtoAExp}$ the Boolean to arithmetic algorithm from [BCZ18]. For Dilithium with Security Level 2, with $q = 2^{23} - 2^{13} + 1$ and $\mu = 18$, we obtain $k = 41$ and $a = 262\,401$.

Algorithm 5 Boolean to Arithmetic conversion ($\mathsf{BtoA_qApprox}$)

Input: A modulus q, a μ-bit Boolean masking $u_1, \ldots, u_n$ such that $u_1 \oplus \cdots \oplus u_n = x$
Output: An arithmetic sharing $y_1, \ldots, y_n$ such that $y_1 + \cdots + y_n = x + e \pmod{q}$, for $0 \leq e < n$.

1: $k \leftarrow \lceil \log_2 q \rceil + \mu$
2: $a \leftarrow \lceil 2^k / q \rceil$
3: $x_1, \ldots, x_n \leftarrow \mathsf{BtoAExp}_{\mu, 2^k}(u_1, \ldots, u_n)$
4: **for** $i = 1$ **to** n **do** $y_i \leftarrow \lfloor x_i \cdot a \cdot q / 2^k \rceil \bmod q$
5: $y_1 \leftarrow y_1 + (n-1) \bmod q$
6: **return** $(y_1, \ldots, y_n)$

Complexity. The values k and a can be computed prior to the execution of the algorithm and are therefore not considered in the 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 loop in Algorithm 5 costs $3n$ operations. Therefore the overall complexity is $T_{\mathsf{BtoA_qApprox}} = 10 \cdot 2^n - 6n - 13 + 3n = 10 \cdot 2^n - 3n - 13$ and asymptotically the number of operation is $\mathcal{O}(2^n)$. As in [BCZ18], this number of operations is exponential in n, but independent from the size of the modulus q.

Lemma 4. *The* $\mathsf{BtoA_qApprox}$ *algorithm is* $(n-1)$-SNI.

Proof. The $\mathsf{BtoA_qApprox}$ algorithm is the composition of the $\mathsf{BtoAExp}$ algorithm which is $(n-1)$-SNI, followed by independent operations on each shares. The resulted algorithm is therefore $(n-1)$-SNI. □

4.3 Exact conversion using modulus switching

The drawback of the previous approach is that we get an error e in the arithmetic masking of x modulo q. In this section, we show how to get rid of the error e, by using the $\mathsf{ShiftMod}$ algorithm from Section 3.1.

As previously, we consider a μ-bit Boolean masking of x with $x = u_1 \oplus \cdots \oplus u_n$, and our goal is to obtain an arithmetic masking of x modulo q, but with no error:

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

As previously, we assume that the bit-size μ and the modulus q are fixed.

We let $\alpha := \lceil \log_2 n \rceil$. Starting from the Boolean shares u_i of x, we consider the shifted shares $u'_i := u_i \ll \alpha$ of $x' = 2^\alpha \cdot x$, of bit-size $\mu' = \mu + \alpha$. We also consider the modulus $q' := 2^\alpha \cdot q$. We apply the previous $\mathsf{BtoA_qApprox}$ algorithm (Alg. 5) with parameters μ' and q', with the shares u'_i as input. This provides an arithmetic masking of $x' = 2^\alpha \cdot x$ modulo q', with an error $0 \leq e \leq n-1$:

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

Since we are working modulo $q' = 2^\alpha \cdot q$, we can then iterate the $\mathsf{ShiftMod}$ algorithm (Alg. 1) α times, and since by definition $0 \leq e \leq n-1 < 2^\alpha$, we get rid of the error e and obtain an arithmetic masking of x modulo q as required:

$$z_1 + \cdots + z_n = \left\lfloor \frac{y_1 + \cdots + y_n}{2^\alpha} \right\rfloor = x \pmod{q}$$

This gives the following $\mathsf{BtoA_qExact}$ algorithm depicted in Algorithm 6.

Algorithm 6 Boolean to Arithmetic conversion ($\mathsf{BtoA_qExact}$)

Input: A modulus q, a μ-bit Boolean masking $x_1, \ldots, x_n$ such that $u_1 \oplus \cdots \oplus u_n = x$
Output: An arithmetic sharing $y_1, \ldots, y_n$ such that $y_1 + \cdots + y_n = x \bmod q$

1: $\alpha \leftarrow \lceil \log_2 n \rceil$, $k \leftarrow \lceil \log_2 q \rceil + \mu + \alpha$, $q' \leftarrow 2^\alpha \cdot q$, $a \leftarrow \lceil 2^k / q \rceil$.
2: $x_1, \ldots, x_n \leftarrow \mathsf{BtoAExp}_{\mu,2^k}(u_1, \ldots, u_n)$
3: **for** $i = 1$ **to** n **do** $y_i \leftarrow \lfloor (x_i \cdot a \cdot q)/2^{k-\alpha} \rfloor \bmod q'$
4: $y_1 \leftarrow y_1 + n - 1$
5: **for** $i = 0$ **to** $\alpha - 1$ **do** $(y_1, \ldots, y_n) \leftarrow \mathsf{ShiftMod}(2^{\alpha-i} \cdot q, (y_1, \ldots, y_n))$
6: **return** $(y_1, \ldots, y_n)$

Note that for simplicity, in the algorithm above, we actually run the initial $\mathsf{BtoAExp}$ algorithm from [BCZ18] directly with the input Boolean shares u_i instead of u'_i, to get

an arithmetic sharing of $x = x_1 + \cdots + x_n \pmod{2^k}$, and then use $x'_i = 2^\alpha \cdot x_i$ for all i, which gives an arithmetic sharing of $x' = 2^\alpha \cdot x$ modulo $2^{k+\alpha}$. Therefore we must have $k' = k + \alpha$, where $k' = \lceil \log_2 q' \rceil + \mu'$ is the parameter used in Alg. 5, which gives $k = k' - \alpha = \lceil \log_2 q \rceil + \mu + \alpha$. We then work with $a = \lceil 2^{k'}/q' \rceil = \lceil 2^k/q \rceil$. The modulus switching is then performed with $y_i = \lfloor (x'_i \cdot a \cdot q')/2^{k'} \rfloor = \lfloor (x_i \cdot a \cdot q)/2^{k-\alpha} \rfloor \bmod q'$.

Complexity. The values α, k, q' and a can be computed prior to the execution of the algorithm and are therefore not considered in the 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 loop at Line 3 costs $3n$ operations. Eventually, the cost of Line 5 is $\alpha(2n^2 + 10n - 9)$ operations, with $\alpha = \lceil \log_2 n \rceil$. Therefore the overall complexity is $T_{\mathsf{BtoA_qExact}} = 10 \cdot 2^n - 6n - 13 + 3n + \alpha(2n^2 + 13n - 9) = 10 \cdot 2^n - 3n - 13 + \alpha(2n^2 + 13n - 9)$ 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.

Lemma 5. *The* $\mathsf{BtoA_qExact}$ *algorithm is* $(n-1)$-NI.

Proof. The $\mathsf{BtoA_qExact}$ algorithm is the composition of the $\mathsf{BtoAExp}$ algorithm which is $(n-1)$-SNI, followed by independent operations on each shares and by the $\mathsf{ShiftMod}$ algorithm which is also $(n-1)$-NI. The resulting algorithm is therefore $(n-1)$-NI. □

4.4 Comparison

We provide a comparison between the various Boolean to arithmetic conversion algorithms in Table 2. We see that for $\mu = 18$ as in Dilithium with Security Level 2, the new algorithms $\mathsf{BtoAqApprox}$ and $\mathsf{BtoAqExact}$ outperform the state of the art algorithms for security order $t \leq 8$. The approximate conversion $\mathsf{BtoAqApprox}$ performs better than $\mathsf{BtoAqExact}$, but with the drawback of a small error in the conversion. However, we show in Appendix C that it can still be safely used in the masking of Dilithium by employing slightly restrictive rejection sampling. While this modification results in perfectly valid Dilithium signatures, the signature algorithm would not fully conform to the standardized algorithm. Therefore, for the high-order masking of Dilithium in sections 6 and 7, we only consider the exact conversion algorithm $\mathsf{BtoAqExact}$. We also show in Appendix D that $\mathsf{BtoAqApprox}$ and $\mathsf{BtoAqExact}$ consume much less randomness than [SPOG19].

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 $q = 2^{23} - 2^{13} + 1$.

$\mathbf{B} \to \mathbf{A} \bmod q$	Security order t							
	2	3	4	5	6	8	10	12
[BBE+18] $18 \to \bmod q$	2 841	5 215	8 782	12 897	17 825	30 235	46 012	64 776
[SPOG19] $18 \to \bmod q$	804	1 414	2 186	3 120	4 216	6 894	10 220	14 194
[CGMZ22] $18 \to \bmod q$	993	1 885	3 065	4 533	6 289	10 665	16 193	22 873
$\mathsf{BtoAqApprox}$	58	135	292	609	1 246	5 080	20 434	81 868
$\mathsf{BtoAqExact}$	154	285	610	1 032	1 786	6 160	21 938	83 860

5 Masking the Decompose function

In Dilithium, the Decompose function is used at signature generation to extract the high and low order bits of $\mathbf{w} = \mathbf{Ay} \bmod q$, with $(\mathbf{w}_0, \mathbf{w}_1) := \mathsf{Decompose}_q(\mathbf{w}, 2\gamma_2)$. We recall the function below, for a single element $r \in \mathbb{Z}_q$.

Algorithm 7 $\mathsf{Decompose}_q(r, \alpha)$

1: $r := r \bmod^+ q$
2: $r_0 := r \bmod^\pm \alpha$
3: **if** $r - r_0 = q - 1$ **then** $r_1 := 0,\ r_0 := r_0 - 1$
4: **else** $r_1 = (r - r_0)/\alpha$
5: **return** (r_1, r_0)

Existing work. In [ABC+22], the authors describe two distinct approaches for masking the $\mathsf{Decompose}_q(r, \alpha)$ function, depending on the value of α. In both cases, the input r is arithmetically masked modulo q, the value r_1 is returned in the clear, while r_0 remains arithmetically masked modulo q. For NIST security level 2, where $\alpha = (q-1)/44$, the authors show that r_1 can be computed using the Compress function from Kyber, which can be masked using a technique described in [CGMZ23]. Although their approach is heuristic, the correctness of their method can be exhaustively verified for all possible values of r modulo q, since q is relatively small. For NIST security levels 3 and 5, with $\alpha = (q-1)/16$, the authors describe a second approach using an arithmetic modulo q to Boolean conversion, taking advantage that 16 is a power-of-two, so that it suffices to keep the 4 least significant bits of the output.

Our contribution. In this section, we present an extension of the two approaches from [ABC+22], so that each approach can cover all NIST security levels. Firstly, we expand on the first approach based on the Compress function, by providing an alternative description of the Decompose procedure used in Dilithium. This alternative method is proven to be equivalent to the original Decompose and can work for any $\alpha = (q-1)/\delta$, not just for $\alpha = (q-1)/44$. We then describe the masking of the resulting algorithm based on our new ShiftMod algorithm, which is more efficient than the full arithmetic to Boolean conversion used in [ABC+22] and [CGMZ23]. The masking complexity of this approach is $\mathcal{O}(n^2 \log qn)$.

Secondly, we extend the second approach of [ABC+22] to cover any $\alpha = (q-1)/\delta$, by providing an alternative description of the Decompose procedure used in Dilithium. This alternative method is straightforward to mask with an arithmetic to Boolean conversion, including NIST security level 2 with $\alpha = (q-1)/44$, whereas the initial approach in [ABC+22] only worked for a power-of-two δ. The masking complexity of this approach is $\mathcal{O}(n^2 \log q)$.

Finally, we provide a comparison of the two approaches. Although the second approach is asymptotically faster, with a complexity of $\mathcal{O}(n^2 \log q)$ instead of $\mathcal{O}(n^2 \log qn)$, the first approach tends to be faster in practice. This is because the second approach requires a full arithmetic modulo q to Boolean conversion, which is more costly than the first approach that can employ our more lightweight ShiftMod algorithm from Section 3.1.

5.1 First alternative Decompose algorithm

Algorithm 7 would be difficult to mask directly, because of the test at Line 3. Therefore, we consider the following alternative algorithm for Decompose, and prove that the two algorithms actually compute the same function. This corresponds to the first approach in [ABC+22], which the authors used for security level 2 with $\alpha = (q-1)/\delta$ where $\delta = 44$. The authors computed the output r_1 using the Compress function from Kyber, which in turn can be masked using a technique described in [CGMZ23]. Although their approach was heuristic, the correctness of their method can be exhaustively verified for all possible values of r modulo q, since q is relatively small in Dilithium.

Using the DecomposeComp algorithm below, we extend this approach to any δ, and prove that the original Decompose and our DecomposeComp algorithms compute the same function. We then provide a more efficient masking technique than in [CGMZ23], based on our ShiftMod algorithm from Section 3.1.

Algorithm 8 $\mathsf{DecomposeComp}_q(r, \alpha)$

1: $\delta := (q-1)/\alpha$
2: $r_1 := \lfloor \delta \cdot r/q \rceil \bmod^+ \delta$
3: $r_0 := r - r_1 \cdot \alpha \bmod^\pm q$
4: **return** (r_1, r_0)

Lemma 6. *Let $\delta \in \mathbb{Z}$ and $\alpha = (q-1)/\delta$, and assume $\alpha = 0 \pmod 2$. The algorithms* Decompose *and* DecomposeComp *compute the same function.*

Proof. We can assume wlog that $0 \leq r < q$. We first consider the original Decompose algorithm, with $(r_1, r_0) \leftarrow \mathsf{Decompose}_q(r, \alpha)$. We write:

$$r = r'_1 \cdot \alpha + r'_0 \tag{5}$$

with $-\alpha/2 < r'_0 \leq \alpha/2$, so that $r'_0 = r \bmod^\pm \alpha$. Therefore r'_0 corresponds to the variable r_0 at Step 2 of Decompose. Note that since $0 \leq r \leq q-1$, we must have $0 \leq r'_1 \leq (q-1)/\alpha = \delta$.

For $0 \leq r < (q-1) - \alpha/2$, we have $0 \leq r'_1 < \delta$, so that $r - r'_0 = r'_1 \cdot \alpha < q-1$, and therefore by definition of Decompose, we have $r_0 = r'_0$ and $r_1 = r'_1$. On the other hand, for $q - 1 - \alpha/2 \leq r < q$, we have $r'_1 = \delta$, which gives $r - r'_0 = q-1$, and therefore $r_1 = 0$ and $r_0 = r'_0 - 1$. Therefore, we always have $r_1 = r'_1 \bmod^+ \delta$.

We now consider $(r''_1, r''_0) \leftarrow \mathsf{DecomposeComp}(r, \alpha)$. We have using (5):

$$\begin{aligned} r''_1 &= \left\lfloor \frac{\delta \cdot r}{q} \right\rceil = \left\lfloor \frac{\delta \cdot (r'_1 \cdot \alpha + r'_0)}{q} \right\rceil \pmod \delta \\ &= \left\lfloor \frac{\delta \cdot r'_0 + (q-1) \cdot r'_1}{q} \right\rceil = r'_1 + \left\lfloor \frac{\delta \cdot r'_0 - r'_1}{q} \right\rceil \pmod \delta \end{aligned} \tag{6}$$

Using $-\alpha/2 < r'_0 \leq \alpha/2$ and $0 \leq r'_1 \leq \delta$, we have $\delta \cdot r'_0 - r'_1 \leq \delta \cdot \alpha/2 \leq (q-1)/2$, and similarly $\delta \cdot r'_0 - r'_1 \geq \delta \cdot (-\alpha/2 + 1) - \delta \geq -(q-1)/2$. Therefore $|\delta \cdot r'_0 - r'_1| \leq (q-1)/2$, which from (6) implies $r''_1 = r'_1 \bmod^+ \delta$ and therefore $r''_1 = r_1$.

Finally, since for both Decompose and DecomposeComp we have $r = r_0 + r_1 \cdot \alpha \pmod q$ and $r = r''_0 + r''_1 \cdot \alpha \pmod q$, and moreover $r_1 = r''_1$, we obtain $r_0 = r''_0 \pmod q$, and from $-(q-1)/2 \leq r_0 \leq (q-1)/2$ and $-(q-1)/2 \leq r''_0 \leq (q-1)/2$, we must have $r''_0 = r_0$. □

Masking DecomposeComp**.** Starting from an arithmetic masking of r modulo q, the main challenge is to mask the computation of $r_1 = \lfloor r \cdot \delta/q \rceil \bmod^+ \delta$ at Step 2. Our approach is captured by Equation (7) in the lemma below. It shows that the high-order computation of $\lfloor x \cdot p/q \rceil$ can be written as the composition of two high-order computations. Firstly we apply a modulus switching from modulo q to modulo $p \cdot 2^\rho$ for a large enough ρ, using the ModSwitch algorithm from Section 4.1. Such algorithm can induce an error $0 \leq e < n$ (see Lemma 2), but since Equation (7) is valid for any such e, we are guaranteed to get the correct result after Euclidean division by 2^ρ. Secondly, from the arithmetic masking modulo $p \cdot 2^\rho$, we perform a sequence of ρ ShiftMod to compute the arithmetic division by 2^ρ, and eventually we get the desired result modulo p. We obtain Algorithm 9 below.

Lemma 7. *For any odd positive integer q, any positive integer p, any $n \geq 2$, any integer $x \in \mathbb{Z}_q$, any integer $0 \leq e < n$ and any integer ρ such that $2^\rho \geq 2q \cdot n$, the following holds:*

$$\left\lfloor \frac{x \cdot p}{q} \right\rceil = \left\lfloor \left(\left\lfloor \frac{x \cdot p \cdot 2^\rho}{q} \right\rceil + 2^{\rho-1} + e \right) / 2^\rho \right\rfloor \pmod p \tag{7}$$

Proof. We can assume $0 \leq x < q$. We write $x \cdot p = y \cdot q + r$ with $-(q-1)/2 \leq r \leq (q-1)/2$, and $y = \lfloor x \cdot p/q \rceil$. This gives:

$$\left\lfloor \frac{x \cdot p \cdot 2^\rho}{q} \right\rfloor + 2^{\rho-1} + e = y \cdot 2^\rho + \left\lfloor \frac{r \cdot 2^\rho}{q} \right\rfloor + 2^{\rho-1} + e$$

Therefore, to get (7), it suffices to ensure that $0 \leq \lfloor r \cdot 2^\rho/q \rfloor + 2^{\rho-1} + e < 2^\rho$. From $2^\rho \geq 2q \cdot n$, we have $2^{\rho-1}/q \geq n$, and therefore $\lceil 2^{\rho-1}/q \rceil \geq n$, which implies using $-(q-1)/2 \leq r \leq (q-1)/2$ and $0 \leq e < n$:

$$\left\lfloor \frac{-(q-1) \cdot 2^{\rho-1}}{q} \right\rfloor + 2^{\rho-1} \leq \left\lfloor \frac{r \cdot 2^\rho}{q} \right\rfloor + 2^{\rho-1} + e < \left\lfloor \frac{(q-1) \cdot 2^{\rho-1}}{q} \right\rfloor + 2^{\rho-1} + n$$

$$0 \leq \left\lfloor \frac{r \cdot 2^\rho}{q} \right\rfloor + 2^{\rho-1} + e < 2^\rho - \left\lceil \frac{2^{\rho-1}}{q} \right\rceil + n \leq 2^\rho$$

□

Algorithm 9 SecDecomposeComp

Input: $x_1, \ldots, x_n \in \mathbb{Z}_q$, with $r = x_1 + \cdots + x_n \bmod q$

Output: r_1 and $y_1, \ldots, y_n \in \mathbb{Z}_q$, such that $(r_1, r_0) = \mathsf{Decompose}_q(r, \alpha)$, with $r_0 = y_1 + \cdots + y_n \pmod q$

1: Let $\rho \leftarrow \lceil \log_2(q \cdot n) \rceil + 1$
2: **for** $i = 1$ **to** n **do** $z_i := \lfloor x_i \cdot \delta \cdot 2^\rho/q \rfloor \bmod (\delta 2^\rho)$
3: $z_1 \leftarrow z_1 + (n-1) + 2^{\rho-1} \bmod (\delta 2^\rho)$
4: **for** $i = 0$ **to** $\rho - 1$ **do** $z_1, \ldots, z_n \leftarrow \mathsf{ShiftMod}(2^{\rho-i} \cdot \delta, (z_1, \ldots, z_n))$
5: $(z_1, \ldots, z_n) \leftarrow \mathsf{RefreshMasks}_\delta(z_1, \ldots, z_n)$
6: $r_1 := \sum_i z_i \bmod^+ \delta$
7: $(y_1, \ldots, y_n) := (x_1, \ldots, x_n)$
8: $y_1 := y_1 - \alpha \cdot r_1 \bmod q$
9: **return** $r_1, (y_1, \ldots, y_n)$

Complexity. To do the masking with DecomposeComp, we first perform the modulus switching with $\delta \cdot 2^\rho$ for $\rho = \lceil \log_2 qn \rceil + 1$. This requires $2n$ operations. Then we perform a sequence of ρ ShiftMod. Then we do a refresh, and compute the sum to recover r_1. The complexity is $T(q,n) = 2n + \rho \cdot T_{\mathsf{ShiftMod}}(n) + T_{\mathsf{Refresh}}(n) + n - 1 + 2 = \mathcal{O}(n^2 \log qn)$.

Security. We claim our algorithm achieves the $t - \mathsf{NI}$ security when the output r_1 is public.

Theorem 4 ($t - \mathrm{NIo}$ of SecDecomposeComp)**.** *For any set of t intermediate variables, there exists a subset I with $|I| \leq t$ such that the t intermediate variables can be perfectly simulated from inputs $x_{|I}$ when r_1 is given to the simulator.*

Proof. Assuming r_1 is public, Algorithm 9 until Line 4 is the composition of the NI gadget ShiftMod with sharewise computations and therefore is NI. Moreover, the shares z_i of r_1 are securely recombined due to the free-SNI of RefreshMasks. □

5.2 Second alternative Decompose algorithm

We consider the following second alternative algorithm for Decompose, and prove that the two algorithms actually compute the same function. This corresponds to the second approach in [ABC+22], which initially worked for $\alpha = (q-1)/\delta$ for a power-of-two δ, as

in NIST security levels 3 and 5 with $\delta = 16$. Using the DecomposeMod algorithm below, we extend this approach to any δ, including $\delta = 44$ as for NIST security level 2.

Algorithm 10 $\mathsf{DecomposeMod}_q(r, \alpha)$

1: $\delta := (q-1)/\alpha$
2: $s := (-\delta \cdot r + (q-1)/2) \bmod^+ q$
3: $r_1 := s \bmod^+ \delta$
4: $r_0 := r - r_1 \cdot \alpha \bmod^\pm q$
5: **return** (r_1, r_0)

Lemma 8. *Let $\delta \in \mathbb{Z}$ and $\alpha = (q-1)/\delta$, and assume $\alpha = 0 \pmod 2$. The algorithms* Decompose *and* DecomposeMod *compute the same function.*

Proof. As in the proof of Lemma 6, we write $r = r'_1 \cdot \alpha + r'_0$ with $-\alpha/2 < r'_0 \leq \alpha/2$. We consider $(r''_1, r''_0) \leftarrow \mathsf{DecomposeMod}(r, \alpha)$. Using $\delta \cdot \alpha = q - 1$, we obtain:

$$s = -\delta \cdot r + (q-1)/2 = -\delta \cdot r'_0 + r'_1 + (q-1)/2 \pmod q$$

From $-\alpha/2 < r'_0 \leq \alpha/2$ and $0 \leq r'_1 \leq \delta$, we must have $0 \leq -\delta \cdot r'_0 + r'_1 + (q-1)/2 \leq q-1$, and therefore $s = -\delta \cdot r'_0 + r'_1 + (q-1)/2$ over $\mathbb{Z}$. From $r''_1 = s \bmod^+ \delta$, we get $r''_1 = r'_1 \bmod^+ \delta$, which implies $r''_1 = r_1$. □

The above DecomposeMod algorithm is relatively easy to mask. Starting from an arithmetic masking of r modulo q, we easily obtain an arithmetic masking of s modulo q at Step 2. Then, to high-order compute $s \bmod^+ \delta$ at Step 3, we first convert the masking of s from arithmetic modulo q to Boolean. One must be careful to use a Boolean masking of a representative of $s \in \mathbb{Z}_q$ such that $0 \leq s < q$, which is the case in the arithmetic to Boolean conversion from [BBE+18]. Then one can convert back from Boolean masking to arithmetic masking modulo δ of $s \bmod^+ \delta$. Using the BtoA_δ algorithm from [SPOG19] which achieves the free-SNI property, one can directly recombine the shares to get r_1 in the clear. Eventually, we obtain an arithmetic sharing modulo q of r_0 simply by subtracting $\alpha \cdot r_1$ to the first share x_1 of the input r. We obtain the SecDecomposeMod algorithm below.

Algorithm 11 SecDecomposeMod

Input: $x_1, \ldots, x_n \in \mathbb{Z}_q$, with $r = x_1 + \cdots + x_n \bmod q$
Output: r_1 and $y_1, \ldots, y_n \in \mathbb{Z}_q$, such that $(r_1, r_0) = \mathsf{Decompose}_q(r, \alpha)$, with $r_0 = y_1 + \cdots + y_n \pmod q$

1: **for** $i = 1$ **to** n **do** $s_i := -\delta \cdot x_i \bmod q$
2: $s_1 := s_1 + (q-1)/2 \bmod q$ ▷ $s = \sum_i s_i \pmod q$
3: $s'_1, \ldots, s'_n := \mathsf{AtoB}_q(s_1, \ldots, s_n)$ ▷ $s'_1 \oplus \cdots \oplus s'_n = s$
4: $s''_1, \ldots, s''_n := \mathsf{BtoA}_\delta(s'_1, \ldots, s'_n)$ ▷ $\sum_i s''_i = s \pmod \delta$
5: $r_1 := \sum_i s''_i \bmod^+ \delta$
6: $(y_1, \ldots, y_n) := (x_1, \ldots, x_n)$
7: $y_1 := y_1 - \alpha \cdot r_1 \bmod q$
8: **return** $r_1, (y_1, \ldots, y_n)$

Remark 1. When δ is a power-of-two, as in NIST security levels 3 and 5, there is no need to perform a Boolean to arithmetic conversion at Step 4, as one can simply recombine the last 4 bits of the Boolean shares s'_i of s after a full mask refreshing, as in [ABC+22].

Complexity. We use the arithmetic modulo q to Boolean conversion AtoB_q from [BBE+18] and the Boolean to arithmetic BtoA_δ conversion from [SPOG19]. The number of operation is therefore:

$$\begin{aligned} T_{\mathsf{SecDecomposeMod}}(n,q) &= n+1+T_{\mathsf{AtoB}}(n,q)+T_{\mathsf{BtoA}}(n,\lceil\log_2 q\rceil)+(n-1)+2 \\ &= \mathcal{O}(\log(q)n^2) \end{aligned}$$

Security. We claim our algorithm achieves the $t-\mathsf{NI}$ security when the output r_1 is public. The proof is similar to the proof of Theorem 4 and is therefore omitted.

Theorem 5 ($t-$ NIo of SecDecomposeMod)**.** *For any set of t intermediate variables, there exists a subset I with $|I| \leq t$ such that the t intermediate variables can be perfectly simulated from inputs $x_{|I}$ when r_1 is given to the simulator.*

5.3 Comparison

In Table 3, we compare different techniques used for masking the Decompose function in the Dilithium signature scheme. Our SecDecomposeComp algorithm is shown to be faster than SecDecomposeMod and [ABC+22] for all security levels. Specifically, for security level 2, our SecDecomposeComp algorithm is faster than [ABC+22] because the latter uses a full arithmetic to Boolean conversion to mask the Compress function, moreover with a non power-of-two modulus, whereas we use our more lightweight ShiftMod algorithm. Likewise, for security levels 3 and 5, our SecDecomposeComp algorithm outperforms [ABC+22], which is equivalent to SecDecomposeMod, as both require a full arithmetic modulo q to Boolean conversion, which is more computationally expensive than our ShiftMod algorithm.

Table 3: Operation count for masking Decompose with $n = t+1$ shares, for prime $q = 2^{23} - 2^{13} + 1$.

		Security order t						
		2	3	4	5	6	8	10
Sec. Level 2	[ABC+22]	2 886	5 365	9 719	14 399	20 040	35 410	54 365
	SecDecomposeComp	1 033	1 669	2 503	3 385	4 378	6 940	9 803
	SecDecomposeMod	3 261	5 952	10 065	14 751	20 359	34 492	52 464
Sec. Level 3 & 5	[ABC+22], SecDecomposeMod	2 196	4 082	7 177	10 632	14 796	25 402	38 995
	SecDecomposeComp	1 033	1 669	2 503	3 385	4 378	6 940	9 803

6 Application to masking Dilithium

Dilithium [BDK+21] is a lattice-based signature scheme based on the MLWE (Module Learning With Errors) and the SelfTargetMSIS (Module Short Integer Solution) problems [LS15]. It was selected by NIST as the primary standard for quantum safe digital signatures. In this section, we describe the complete high-order masking of Dilithium, utilizing the improved gadgets described in the previous sections. We provide the practical results of a C implementation and compare the performance of our new gadgets with those from [BBE+18], [SPOG19], and [ABC+22]. For simplicity, we focus on the randomized version of Dilithium, since as shown in [ABC+22] it leads to a significantly faster masked implementation than the deterministic version. We recall in Table 4 the parameters of Dilithium for NIST security levels 2, 3 and 5.

Table 4: Dilithium parameters for different NIST security levels

Level	q	γ_1	γ_2	(k,l)	η	β	ω	Repetitions
2	8380417	2^{17}	$(q-1)/88$	$(4,4)$	2	78	80	4.25
3	8380417	2^{19}	$(q-1)/32$	$(6,5)$	4	196	55	5.1
5	8380417	2^{19}	$(q-1)/32$	$(8,7)$	2	120	75	3.85

6.1 Pseudo-code of Dilithium

We first recall the pseudo-code of Dilithium in Fig. 1, using the version from the reference implementation, and formally described in [ABC+22]. Namely, one can distinguish two equivalent versions of Dilithium signatures, the $\mathbf{r}$-version and the $\tilde{\mathbf{r}}$-version [ABC+22]. The Dilithium specification [BDK+21, Fig. 4] provides the pseudo-code description of the $\mathbf{r}$-version. In this version, one computes $\mathbf{w}_1 := \mathsf{HighBits}_q(\mathbf{w}, 2\gamma_2)$ and $\mathbf{r}_0 := \mathsf{LowBits}_q(\mathbf{w} - c\mathbf{s}_2, 2\gamma)$, and performs the rejection sampling on $\mathbf{r}_0$. On the other hand, the $\tilde{\mathbf{r}}$-version comes from the reference implementation, as discussed in [BDK+21, Section 5.1], and formally described in [ABC+22, Appendix A]. We recall its pseudo-code description in Fig. 1.

In this $\tilde{\mathbf{r}}$-version, one first computes $(\mathbf{w}_0, \mathbf{w}_1) \leftarrow \mathsf{Decompose}_q(\mathbf{w}, 2\gamma_2)$ at Line 5, and then computes $\tilde{\mathbf{r}} := \mathbf{w}_0 - c\mathbf{s}_2$ at Line 9. Finally, one performs the rejection sampling on $\tilde{\mathbf{r}}$ at Line 10. In the rest of this paper, as in [ABC+22], we consider this $\tilde{\mathbf{r}}$-version because it is easier to mask. We refer to [BDK+21] for the definition of the $\mathsf{UseHint}$ and $\mathsf{MakeHint}$ algorithms. Given $(\mathbf{w}_0, \mathbf{w}_1) \leftarrow \mathsf{Decompose}_q(\mathbf{w}, 2\gamma_2)$, as in [BDK+21] we write $\mathsf{HighBits}_q(\mathbf{w}, 2\gamma_2) := \mathbf{w}_1$ and $\mathsf{LowBits}_q(\mathbf{w}, 2\gamma_2) := \mathbf{w}_0$.

We now show that signature verification works. Using

$$\mathbf{Az} - c\mathbf{t} = \mathbf{A}(\mathbf{y} + c\mathbf{s}_1) - c(\mathbf{As}_1 + \mathbf{s}_2) = \mathbf{Ay} - c\mathbf{s}_2$$

and from $\mathbf{w} = \mathbf{Ay} = \mathbf{w}_1 \cdot \alpha + \mathbf{w}_0 \pmod q$, we have:

$$\begin{aligned}\mathbf{h} &= \mathsf{MakeHint}_q(-c\mathbf{t}_0, \mathbf{w}_0 - c\mathbf{s}_2 + \alpha \cdot \mathbf{w}_1 + c\mathbf{t}_0, 2\gamma_2)\\ &= \mathsf{MakeHint}_q(-c\mathbf{t}_0, \mathbf{w} - c\mathbf{s}_2 + c\mathbf{t}_0, 2\gamma_2)\\ &= \mathsf{MakeHint}_q(-c\mathbf{t}_0, \mathbf{Az} - c\mathbf{t} + c\mathbf{t}_0, 2\gamma_2) \qquad (8)\end{aligned}$$

Using $\mathbf{Az} - c\mathbf{t}_1 \cdot 2^d = \mathbf{Az} - c\mathbf{t} + c\mathbf{t}_0$, we have $\mathbf{w}'_1 = \mathsf{UseHint}_q(\mathbf{h}, \mathbf{Az} - c\mathbf{t} + c\mathbf{t}_0, 2\gamma_2)$. From (8) and using $\|c\mathbf{t}_0\|_\infty < \gamma_2$, this enables to recover the high bits of $\mathbf{Az} - c\mathbf{t}$ from the hint $\mathbf{h}$, using [BDK+21, Lemma 1]:

$$\begin{aligned}\mathbf{w}'_1 &= \mathsf{UseHint}_q(\mathbf{h}, \mathbf{Az} - c\mathbf{t} + c\mathbf{t}_0, 2\gamma_2)\\ &= \mathsf{HighBits}_q(\mathbf{Az} - c\mathbf{t}, 2\gamma_2) = \mathsf{HighBits}_q(\mathbf{w} - c\mathbf{s}_2, 2\gamma_2)\end{aligned}$$

Moreover, using [BDK+21, Lemma 3], from $\|\mathbf{w}_0 - c\mathbf{s}_2\|_\infty < \gamma_2 - \beta$, we have $\mathsf{HighBits}_q(\mathbf{w} - c\mathbf{s}_2, 2\gamma_2) = \mathsf{HighBits}_q(\mathbf{w}, 2\gamma_2)$ and therefore $\mathbf{w}'_1 = \mathbf{w}_1$, which gives $\tilde{c} = H(M \,\|\, \mathbf{w}'_1)$ as required.

KeyGen

$\mathbf{A} \leftarrow R_q^{k \times \ell}$
$(\mathbf{s}_1, \mathbf{s}_2) \leftarrow S_\eta^\ell \times S_\eta^k$
$\mathbf{t} := \mathbf{A}\mathbf{s}_1 + \mathbf{s}_2$
$(\mathbf{t}_1, \mathbf{t}_0) := \mathsf{Power2Round}_q(\mathbf{t}, d)$
return $(pk = (\mathbf{A}, \mathbf{t}_1),\ sk = (\mathbf{A}, \mathbf{t}_1, \mathbf{s}_1, \mathbf{s}_2, \mathbf{t}_0))$

$\mathsf{Sign}(sk, M)$

$(\mathbf{z}, \mathbf{h}) := \perp$
while $(\mathbf{z}, \mathbf{h}) = \perp$ **do**
    $\mathbf{y} \leftarrow \tilde{S}_{\gamma_1}^\ell$
    $\mathbf{w} := \mathbf{A}\mathbf{y}$
    $(\mathbf{w}_0, \mathbf{w}_1) := \mathsf{Decompose}_q(\mathbf{w}, 2\gamma_2)$
    $\tilde{c} := H(M \,\|\, \mathbf{w}_1)$
    $c := \mathsf{SampleInBall}(\tilde{c})$
    $\mathbf{z} := \mathbf{y} + c\mathbf{s}_1$
    $\tilde{\mathbf{r}} := \mathbf{w}_0 - c\mathbf{s}_2$
    if $\|\mathbf{z}\|_\infty \geq \gamma_1 - \beta$ or $\|\tilde{\mathbf{r}}\|_\infty \geq \gamma_2 - \beta$, then $(\mathbf{z}, \mathbf{h}) := \perp$
    else
        $\mathbf{h} := \mathsf{MakeHint}_q(-c\mathbf{t}_0, \tilde{\mathbf{r}} + \alpha \cdot \mathbf{w}_1 + c\mathbf{t}_0, 2\gamma_2)$
        if $\|c\mathbf{t}_0\|_\infty \geq \gamma_2$ **or** the # of 1's in $\mathbf{h}$ is greater than ω, then $(\mathbf{z}, \mathbf{h}) := \perp$
end while
return $\sigma = (\tilde{c}, \mathbf{z}, \mathbf{h})$

$\mathsf{Verify}(pk, M, \sigma = (\tilde{c}, \mathbf{z}, \mathbf{h}))$

$c := \mathsf{SampleInBall}(\tilde{c})$
$\mathbf{w}_1' := \mathsf{UseHint}_q(\mathbf{h}, \mathbf{A}\mathbf{z} - c\mathbf{t}_1 \cdot 2^d, 2\gamma_2)$
return $[\![\|\mathbf{z}\|_\infty < \gamma_1 - \beta]\!]$ **and** $[\![\tilde{c} = H(M \,\|\, \mathbf{w}_1')]\!]$ **and** $[\![\#$ of 1's in $\mathbf{h}$ is $\leq \omega]\!]$

Figure 1: Template of Dilithium with $\tilde{\mathbf{r}}$-version (reference implementation).

6.2 Masking Dilithium key generation

In the key generation of Dilithium, the private-key elements $\mathbf{s}_1$ and $\mathbf{s}_2$ must be returned as arithmetically masked modulo q. As shown in Figure 1, the variables $\mathbf{s}_1$ and $\mathbf{s}_2$ are sampled uniformly from S_η^ℓ and S_η^k respectively, *i.e*, vectors of polynomials with each coefficient uniformly and independently distributed in $[-\eta, \eta]$. To obtain a masked generation, since η is a small value (e.g., $\eta = 2$, 4, and 2 for NIST security levels 2, 3, and 5, respectively), we proceed by first generating n arithmetic shares $\mathbf{s}_{1,i}$ and $\mathbf{s}_{2,i}$ uniformly in the range $[0, 2\eta + 1[$. This corresponds to an arithmetic sharing modulo $2\eta + 1$ of $\mathbf{s}_1$ and $\mathbf{s}_2$.

Next, we convert this arithmetic sharing modulo $2\eta + 1$ into an arithmetic masking modulo q using the table-based countermeasure from [CGMZ22], which has a complexity of $\mathcal{O}(n^2)$. Specifically, the table has $2\eta + 1$ rows for $-\eta \leq i \leq \eta$, and initially, the row of index i is an encoding of i modulo q. The rows of the table are randomly rotated and refreshed n times, resulting in an n-encoding modulo q of a uniform random in $[-\eta, \eta]$. The same operation is repeated for each coefficient of $\mathbf{s}_1$ and $\mathbf{s}_2$.

With the arithmetic sharing modulo q of $\mathbf{s}_1$ and $\mathbf{s}_2$, we can then high-order compute $\mathbf{t} = \mathbf{A}\mathbf{s}_1 + \mathbf{s}_2$ using the linearity over $\mathbb{Z}_q$. Since the security proof of Dilithium considers $\mathbf{t}$ to be public [BDK+21], we can securely recombine the shares by applying a full refresh. The overall complexity of the masked key-generation algorithm is $\mathcal{O}(n^2)$, as summarized in Figure 2.

KeyGen

$\mathbf{A} \leftarrow R_q^{k \times \ell}$
for $i = 1$ to n **do**
  $\mathbf{s}_{1,i} \leftarrow [0, 2\eta + 1[^{256\ell}$, $\mathbf{s}_{2,i} \leftarrow [0, 2\eta + 1[^{256k}$ ▷ $\mathbf{s}_j = \sum_i \mathbf{s}_{j,i} \bmod^{\pm} 2\eta + 1$
end for
$\mathbf{s}_{1,1}, \ldots, \mathbf{s}_{1,n} \leftarrow \mathsf{A}_{2\eta+1}\mathsf{toA_q Table}(\mathbf{s}_{1,1}, \ldots, \mathbf{s}_{1,n})$
$\mathbf{s}_{2,1}, \ldots, \mathbf{s}_{2,n} \leftarrow \mathsf{A}_{2\eta+1}\mathsf{toA_q Table}(\mathbf{s}_{2,1}, \ldots, \mathbf{s}_{2,n})$
for $i = 1$ **to** n **do** $\mathbf{t}_i := \mathbf{A}\mathbf{s}_{1,i} + \mathbf{s}_{2,i} \bmod q$
$\mathbf{t}_1, \ldots, \mathbf{t}_n = \mathsf{Refresh}(\mathbf{t}_1, \ldots, \mathbf{t}_n)$
$\mathbf{t} := \sum_i \mathbf{t}_i \bmod q$
$(\mathbf{t}_1, \mathbf{t}_0) := \mathsf{Power2Round}_q(\mathbf{t}, d)$
return $(pk = (\mathbf{A}, \mathbf{t}_1),\ sk = (\mathbf{A}, \mathbf{t}_1, (\mathbf{s}_{1,1}, \ldots, \mathbf{s}_{1,n}), (\mathbf{s}_{2,1}, \ldots, \mathbf{s}_{2,n}), \mathbf{t}_0))$

Figure 2: Masked Dilithium key generation

6.3 Masking Dilithium signature generation

Our masking strategy for Dilithium signature generation is similar to that of [ABC+22]. Specifically, we mask the same variables and leave unmasked the same variables. The first step is to mask the random vector of polynomials $\mathbf{y}$, since its knowledge would directly lead to the computation of $\mathbf{s}_1 = (\mathbf{z} - \mathbf{y})/c$, given that $\mathbf{z}$ and c are public. Similarly, we need to mask $\mathbf{w} = \mathbf{A}\mathbf{y}$, as $\mathbf{A}$ is full rank with high probability and could allow one to retrieve $\mathbf{y}$ from $\mathbf{w}$.

As explained in introduction, following [ABC+22], the variable $\mathbf{w}_1$ in $(\mathbf{w}_1, \mathbf{w}_0) \leftarrow \mathsf{Decompose}_q(\mathbf{w})$ need not be masked since it is also publicly computed during signature verification. Thus, the challenge $\tilde{c} = H(M \| \mathbf{w}_1)$ need not be masked, and the Keccak hash function H need not be masked. However, $\mathbf{w}_0$ must remain masked, as using $\mathbf{w}_1$ would allow one to recover $\mathbf{w} = \mathbf{w}_1 \cdot \alpha + \mathbf{w}_0 \bmod q$ and eventually $\mathbf{y}$ and $\mathbf{s}_1$.

We stress that the variables $\mathbf{z}$ and $\tilde{\mathbf{r}}$ must remain masked until the rejection sampling has been performed. If the value $\mathbf{z}$ is rejected due to an out-of-bounds coefficient $|z_i| \geq \gamma_1 - \beta$, then the knowledge of z_i would leak information about the secret $\mathbf{s}_1$. The same holds for $\tilde{\mathbf{r}}$. Therefore, the variables $\mathbf{z}$ and $\tilde{\mathbf{r}}$ must remain masked until the rejection sampling has been completely performed, and the rejection sampling must be done in a masked way so that whenever a polynomial coefficient is rejected, the adversary does not learn more about the rejected coefficient.

Lastly, after the rejection sampling of $\mathbf{z}$ and $\tilde{\mathbf{r}}$, one can then securely recombine the shares of $\mathbf{z}$. Namely, without the public-key compression, $\mathbf{z}$ would have been returned as a valid Dilithium signature. Note that we cannot recombine the shares of $\mathbf{z}$ before the rejection sampling of $\tilde{\mathbf{r}}$, as knowing $\mathbf{z}$ would enable to compute:

$$\tilde{\mathbf{r}} = \mathbf{w}_0 - c\mathbf{s}_2 = \mathbf{w} - \alpha\mathbf{w}_1 - c\mathbf{s}_2 = \mathbf{A}\mathbf{z} - c\mathbf{t} - \alpha\mathbf{w}_1$$

and assuming heuristically that the coefficients of $\mathbf{w}_0$ are uniformly distributed in $]-\gamma_2, \gamma_2]$, an out-of-bound coefficient of $\tilde{\mathbf{r}}$ would leak information about $\mathbf{s}_2$. Eventually, after the rejection sampling of $\mathbf{z}$ and $\tilde{\mathbf{r}}$, once the shares of $\mathbf{z}$ have been recombined, as in [ABC+22] the hint $\mathbf{h}$ can be computed in the clear, by computing $\mathbf{h} = \mathsf{MakeHint}(-c\mathbf{t}_0, \mathbf{A}\mathbf{z} - c\mathbf{t} + c\mathbf{t}_0, 2\gamma_2)$.

In the following, we describe in more details the high-order masking of Dilithium signature generation, following each step of Fig. 1. The globally masked algorithm is formally described in Figure 3 below.

Generation of $\mathbf{y} \in \tilde{S}^{\ell}_{\gamma_1}$. In order to generate $\mathbf{y} \in \tilde{S}^{\ell}_{\gamma_1}$ at Step 3 in Fig. 1, we first generate an n-shared Boolean masking of each coefficient of $\mathbf{y}$, which is then converted into an arithmetic masking using the technique described in Section 4.3.

$\underline{\mathsf{Sign}(sk, M)}$

$(\mathbf{z}, \mathbf{h}) := \perp$
while $(\mathbf{z}, \mathbf{h}) = \perp$ **do**
  for $i = 1$ **to** n **do** $\mathbf{y}_i \leftarrow [0, 2\gamma_1[^{256\ell}$
  $\mathbf{y}_1, \ldots, \mathbf{y}_n := \mathsf{BtoA_q Exact}(\mathbf{y}_1, \ldots, \mathbf{y}_n)$
  $\mathbf{y}_1 := \mathbf{y}_1 - (\gamma_1 - 1)$ ▷ $\mathbf{y} = \sum_i \mathbf{y}_i \bmod^{\pm} q$
  for $i = 1$ **to** n **do** $\mathbf{W}_i := \mathbf{A}\mathbf{y}_i$ ▷ $\mathbf{w} = \sum_i \mathbf{W}_i \pmod q$
  $\mathbf{w}_1, (\mathbf{w}_{0,1}, \ldots, \mathbf{w}_{0,n}) := \mathsf{SecDecompose}((\mathbf{W}_1, \ldots, \mathbf{W}_n), 2\gamma_2)$ ▷ $\mathbf{w}_0 = \sum_i \mathbf{w}_{0,i} \pmod q$
  $\tilde{c} := H(M \,\|\, \mathbf{w}_1)$
  $c := \mathsf{SampleInBall}(\tilde{c})$
  for $i = 1$ **to** n **do** $\mathbf{z}_i := \mathbf{y}_i + c\mathbf{s}_{1,i}$ ▷ $\mathbf{z} = \sum_i \mathbf{z}_i \pmod q$
  if $\mathsf{SecRejectionSampling}((\mathbf{z}_1, \ldots, \mathbf{z}_n), \gamma_1, \beta)$ **then** $(\mathbf{z}, \mathbf{h}) = \perp$
  else
    for $i = 1$ **to** n **do** $\tilde{\mathbf{r}}_i := \mathbf{w}_{0,i} - c\mathbf{s}_{2,i} \bmod q$ ▷ $\tilde{\mathbf{r}} = \sum_i \tilde{\mathbf{r}}_i \pmod q$
    if $\mathsf{SecRejectionSampling}((\tilde{\mathbf{r}}_1, \ldots, \tilde{\mathbf{r}}_n), \gamma_2, \beta)$, **then** $(\mathbf{z}, \mathbf{h}) := \perp$
    else
      $\mathbf{z}_1, \ldots, \mathbf{z}_n := \mathsf{RefreshMasks}(\mathbf{z}_1, \ldots, \mathbf{z}_n)$
      $\mathbf{z} = \sum_i \mathbf{z}_i \bmod q$
      $\mathbf{r} := \mathbf{A}\mathbf{z} - c\mathbf{t} \bmod q$
      $\mathbf{h} := \mathsf{MakeHint}_q(-c\mathbf{t}_0, \mathbf{r} + c\mathbf{t}_0, 2\gamma_2)$
      if $\|c\mathbf{t}_0\|_\infty \geq \gamma_2$ **or** the # of 1's in $\mathbf{h}$ is greater than ω, then $(\mathbf{z}, \mathbf{h}) := \perp$
end while
return $\sigma = (\tilde{c}, \mathbf{z}, \mathbf{h})$

Figure 3: Template of Masked Dilithium signature generation.

More precisely, in the signature generation of Dilithium, each coefficient of $\mathbf{y}$ must be uniformly distributed in $]-\gamma_1, \gamma_1]$. For this, we can first generate each share uniformly in $[0, 2\gamma_1[$ and then subtract $\gamma_1 - 1$. Since γ_1 is a power-of-two ($\gamma_1 = 2^{17}$, 2^{19}, 2^{19} for levels 2, 3, 5 respectively), the uniform generation in $[0, 2\gamma_1[$ corresponds to the uniform generation of a $(\log_2 \gamma_1 + 1)$-bit integer, so we generate the n Boolean shares as random $(\log_2 \gamma_1 + 1)$-bit integers.

We then convert the Boolean masking into an arithmetic masking modulo q, using the exact conversion method described in Section 4.3. This gives an arithmetic sharing modulo q of $\mathbf{y} \in [0, 2\gamma_1[^{256\ell}$ that can eventually be translated into $]-\gamma_1, \gamma_1]^{256\ell}$ by subtracting $\gamma_1 - 1$ to each coefficient of the first share. We obtain an arithmetic masking $\mathbf{y}_1, \ldots, \mathbf{y}_n$ modulo q of the signing nonce $\mathbf{y}$.

Computation of $\mathbf{w} = \mathbf{A}\mathbf{y}$. From the linearity of matrix multiplication in $\mathbb{Z}_q$, the vector $\mathbf{w}$ can be computed share by share. In Fig. 3, we denote by $\mathbf{W}_i = \mathbf{A} \cdot \mathbf{y}_i \bmod q$ the arithmetic shares of $\mathbf{w}$ modulo q. As explained above, we cannot unmask the shares $\mathbf{W}_i$ into $\mathbf{w}$, as this would lead to $\mathbf{y}$ since $\mathbf{A}$ is public and full rank with high probability.

Computation of $(\mathbf{w_0}, \mathbf{w_1}) = \mathsf{Decompose}_q(\mathbf{w}, \mathbf{2\gamma_2})$. As explained previously, we can compute $\mathbf{w}_1$ in the clear, but the Decompose function must be high-order computed, and $\mathbf{w}_0$ must remain arithmetically masked, as in [ABC+22]. Such masking of Decompose is described in Section 5.2, with two possible approaches. In Fig. 3, we denote by $\mathbf{w}_{0,i}$ the n arithmetic shares modulo q of $\mathbf{w}_0$.

Computation of $\mathbf{z}$ and $\tilde{\mathbf{r}}$. The vector $\mathbf{z}$ is computed share by share from the shares $\mathbf{y}_i$ and $\mathbf{s}_{1,i}$ of $\mathbf{y}$ and $\mathbf{s}_1$ respectively. The same holds for $\tilde{\mathbf{r}} = \mathbf{w}_0 - c\mathbf{s}_2$.

Rejection sampling of $\mathbf{z}$ and $\tilde{\mathbf{r}}$. We implement the rejection sampling of $\mathbf{z}$ and $\tilde{\mathbf{r}}$ using the method described in [ABC+22]. The test is conducted on the arithmetic shares $\mathbf{z}_i$ of $\mathbf{z}$ and the arithmetic shares $\tilde{\mathbf{r}}_i$ of $\tilde{\mathbf{r}}$. We perform the rejection sampling iteratively on the masked coefficients of $\mathbf{z}$ and $\tilde{\mathbf{r}}$, and the result of the test is computed in the clear for each coefficient. If any coefficient fails the test, the vectors $\mathbf{z}$ and $\tilde{\mathbf{r}}$ are rejected immediately. To perform the masked rejection sampling of each coefficient, we utilize the method from [ABC+22] based on the SecA2BModp and SecLeq gadgets. This method is more efficient than the technique presented in [BBE+18].

Recombining the shares of $\mathbf{z}$ and computing the hint. As explained above, after the rejection sampling of $\mathbf{z}$ and $\tilde{\mathbf{r}}$, one can securely recombine the shares of $\mathbf{z}$, after a full mask refreshing, and then compute $\mathbf{h} = \mathsf{MakeHint}(-c\mathbf{t}_0, \mathbf{Az} - c\mathbf{t} + c\mathbf{t}_0, 2\gamma_2)$ in the clear. We summarize the fully masked Dilithium signature generation in Fig. 3 above.

Complexity. Based on the individual complexities of the gadgets, we can conclude that the overall complexity of signature generation is $\mathcal{O}(2^n + n^2 \log q)$ when using the $\mathsf{BtoA_q Exact}$ algorithm from Section 4 for generating $\mathbf{y}$, and the SecDecomposeComp algorithm from Section 5.1. As shown in Section 4.4, our $\mathsf{BtoA_q Exact}$ algorithm has exponential complexity $\mathcal{O}(2^n)$, but is much faster than other algorithms for small values of n. To guarantee a polynomial-time complexity in n, one can use the algorithm proposed in [SPOG19] instead. In that case, the overall complexity of signature generation reduces to $\mathcal{O}(n^2 \log q)$.

Security. Given that each gadget processing sensitive variable achieves at least $t - \mathsf{NI}$ security and the public variables are securely recombined, the whole signing procedure achieves, by composition, t-NIo security when the output signature σ is public.

Theorem 6 ($t-$NIo security of Sign)**.** *For any set of t intermediate variables, there exists a subset $I \subset [1, n]$, with $|I| \leq t$, such that these t intermediate variables can be perfectly simulated from $\mathbf{s}_{1,|I}$ and $\mathbf{s}_{2,|I}$ when the output σ is given to the simulator.*

7 Implementation results

We have implemented the gadgets introduced in the previous sections in C and integrated them into the reference implementation of Dilithium, resulting in a fully masked implementation of Dilithium in C. We provide below the benchmark results obtained on a laptop with an `Intel(R) Core(TM) i7-1065G7 @1.30GHz` CPU. We observe that the performance of each gadget is similar to the operation count provided in the previous sections. However, we emphasize that our implementation is only a proof-of-concept and the actual performance in a real-life product would depend on the specific device targeted. To achieve optimal performance, an optimized implementation of the gadgets for the targeted architecture and mitigation of micro-architectural leakage would be necessary, but these steps are beyond the scope of this work. The plain C code is publicly available at

https://github.com/fragerar/Masked_Dilithium

7.1 Performance of individual gadgets

We provide below the performance results of individual gadgets. The randomness used is fetched from a simple `XorShift` PRNG. In practice, this would need to be replaced by a TRNG in a secure environment.

Table 5: Cycle counts for a full arithmetic to Boolean conversion on 32-bit masked values.

A mod 2^{32} → B	Security order t								
	1	2	3	4	5	6	7	8	9
[Gou01]/[CGV14]	50	1067	1649	3225	5176	6324	7392	10360	13248
Alg. 3 with ShiftMod	1262	2099	3179	4339	5375	6404	7655	9099	10650

Table 6: Cycle counts for a μ-bit Boolean to arithmetic mod q conversion for $\mu = 18$ and $q = 2^{23} - 2^{13} + 1$.

		Security order t					
		1	2	3	4	5	6
B → A mod 2^k	B2Amod2^k [BCZ18]	23	79	308	516	906	1961
B → A mod q	[SPOG19]	694	1307	2275	3395	4288	6251
	BtoAqApprox	24	80	310	518	908	1963
	BtoAqExact	351	445	979	1270	2133	3537

Arithmetic mod 2^k to Boolean conversion. Table 5 presents the performance of the arithmetic mod 2^k to Boolean conversion using the new ShiftMod gadget proposed in Section 3.1. The table shows that the empirical results are consistent with the theoretical operation counts in Table 1, although in practice, ShiftMod outperforms [CGV14] for orders slightly higher than expected.

Boolean to arithmetic mod q conversion. Table 6 compares various approaches for converting Boolean to arithmetic mod q to generate the coefficients of $\mathbf{y}$ within a given range, showing similar results to the operation counts presented in Table 2. The table shows that the newly proposed BtoAqApprox and BtoAqExact algorithms outperform the previously proposed algorithm in [SPOG19] for small orders. However, for higher orders, the main bottleneck arises from the Boolean to arithmetic conversion modulo 2^k proposed in [BCZ18], which has exponential complexity. As expected, the overhead for BtoAqApprox is lower than that for BtoAqExact.

Comparison with the bitsliced approach. We provide a high-level comparison with the bitsliced approach of [BC22] for the high-order Boolean to arithmetic conversion modulo q. For such conversion, the authors of [BC22] provide a comparison between their bitsliced implementation of the Boolean to arithmetic conversion algorithm from [BBE+18], and the non-bitsliced implementation of the [SPOG19] conversion algorithm. We denote by k the bitsize of the modulus q, and by μ the input Boolean size. The complexity of [BC22] is $\mathcal{O}(k)$ and independent of μ, while the complexity of [SPOG19] is $\mathcal{O}(\mu)$ and independent of k. In [BC22, Fig. 6], for $k = 12$ and $\mu = 1$, bitsliced [BC22] is 2.5 as slow as non-bitsliced [SPOG19], for security order $t = 2$. Based on this, we can estimate that for $k = 23$ and $\mu = 18$, as in Dilithium, bitsliced [BC22] would be 3.7 times faster than non-bitsliced [SPOG19]. On the other hand, from Table 6 above, we observe that for security order $t = 2$, our non-bitsliced implementation is 2.9 times faster than [SPOG19]. Therefore, for the Boolean to arithmetic modulo q conversion, we expect [BC22] to be 1.3 times faster than our non-bitsliced implementation (and 2.4 times faster for security order $t = 6$).

However, the timing estimates above do not take into account the change of representation required for a bitsliced implementation, from the canonical representation where the inputs bits are stored contiguously in a given register, to the bitsliced representation where each input bit is stored in a different register for parallel evaluation, and vice versa (see Section 2.6 in [BC22]). While this change of representation has a linear complexity in

Table 7: Cycle counts for the Decompose gadgets, for security level 2 with $\gamma_2 = (q-1)/88$, and security levels 3 and 5 with $\gamma_2 = (q-1)/32$.

		Security order t					
		1	2	3	4	5	6
$\gamma_2 = (q-1)/88$	DecomposeComp	1814	2967	5425	8468	10859	13847
	DecomposeMod	2280	7594	12862	20514	25689	34012
$\gamma_2 = (q-1)/32$	DecomposeComp	1660	2459	3619	5171	7348	8912
	DecomposeMod ([ABC+22])	1173	2787	4662	7285	10345	13602

the number of shares, it can impact the overall efficiency of a bitsliced approach for the full Dilithium algorithm.

Masking of Decompose. Table 7 compares the two methods proposed in Section 5.2 for performing the `Decompose` operation. As expected, the `DecomposeComp` approach is more efficient than the alternative `DecomposeMod` approach, since it only requires a sequence of arithmetic shifts using the ShiftMod operation instead of a full arithmetic mod q to Boolean conversion. However, when $\gamma_2 = (q-1)/32$, which applies to security levels 3 and 5, the difference in efficiency between the two approaches is slightly smaller because the Boolean to arithmetic conversion can be avoided in Algorithm 11. We also provide in Appendix D the randomness consumption of the main gadgets.

7.2 Performance of fully-masked Dilithium

Table 8 presents the cycle counts for the full signature generation at security levels 2, 3, and 5, along with the time spent in each signature component for security level 3. The running time corresponds to the average total time spent in each component during multiple executions of the signature generation, including restarts due to rejection sampling.

The table highlights a common feature of high-order implementations of lattice-based schemes that were not initially designed with masking in mind: the slowest operations are often the ones that are trivial to compute in the unmasked case. We can see that the majority of the time is spent in `Decompose` and `Reject`, which are simple operations in the absence of masking. In contrast, polynomial arithmetic, which is the primary optimization target for unprotected implementations, only takes a small fraction of the runtime in masked implementations as it is a linear operation.

Table 8: Cycle counts for a full implementation of randomized Dilithium, with a breakdown of the time taken by each signature component for security level 3. Values are expressed in thousands of cycles. Average over 1000 executions of the signature at each order.

	Security order t						
	0	1	2	3	4	5	6
Dilithium2	506	26602 (×53)	54945 (×109)	101020 (×200)	155025 (×306)	231265 (×457)	296200 (×585)
Dilithium3	853	36986 (×43)	83696 (×98)	130590 (×153)	205473 (×241)	273446 (×321)	395518 (×464)
Dilithium5	989	38069 (×38)	87809 (×89)	137034 (×139)	201838 (×204)	315249 (×319)	404769 (×409)
NTTs	-	304	451	598	732	838	1037
Sample y	-	3034	5135	7890	13127	19805	35884
Compute Ay	-	616	916	1121	1515	1704	2182
Decompose	-	13088	32963	52030	84131	110119	158275
$z = y + c \cdot s_1$	-	355	528	641	872	981	1245
Reject	-	18956	42856	67281	103840	138584	195207
$w - c \cdot s_2$	-	262	390	487	635	746	932

8 Conclusion

In this work, we presented improved high-order gadgets for the masking of Dilithium, a post-quantum lattice-based signature scheme standardized by NIST. Our new gadget called ShiftMod enables efficient arithmetic shifts modulo any integer $2q$, which we used as a component in other masking gadgets. We also proposed a fast Boolean-to-arithmetic modulo q conversion algorithm that leverages ShiftMod, which is used in Dilithium for masking the generation of the random variable y modulo q.

Additionally, we described improved techniques for masking the Decompose function in Dilithium, and we demonstrated the effectiveness of our countermeasures with a complete high-order masked implementation of Dilithium. We provided practical results of a C implementation and compared the performance improvement provided by the new gadgets with those from previous work.

References

[ABC+22] Melissa Azouaoui, Olivier Bronchain, Gaëtan Cassiers, Clément Hoffmann, Yulia Kuzovkova, Joost Renes, Markus Schönauer, Tobias Schneider, François-Xavier Standaert, and Christine van Vredendaal. Leveling dilithium against leakage: Revisited sensitivity analysis and improved implementations. Cryptology ePrint Archive, Paper 2022/1406, 2022. 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 - 24th International Conference on the Theory and Application of Cryptology and Information Security, Brisbane,*

QLD, Australia, December 2-6, 2018, Proceedings, Part I, volume 11272 of *Lecture Notes in Computer Science*, pages 494–524. Springer, 2018.

[BDH+21] Shivam Bhasin, Jan-Pieter D'Anvers, Daniel Heinz, Thomas Pöppelmann, and Michiel Van Beirendonck. Attacking and defending masked polynomial comparison for lattice-based cryptography. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2021(3):334–359, 2021. https://eprint.iacr.org/2021/104.

[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.

[BG14] Shi Bai and Steven D. Galbraith. An improved compression technique for signatures based on learning with errors. In Josh Benaloh, editor, *Topics in Cryptology – CT-RSA 2014*, pages 28–47, Cham, 2014. Springer International Publishing.

[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.

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

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

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

[CS21] Jean-Sébastien Coron and Lorenzo Spignoli. Secure wire shuffling in the probing model. In Tal Malkin and Chris Peikert, editors, *Advances in Cryptology - CRYPTO 2021 - 41st Annual International Cryptology Conference, CRYPTO 2021, Virtual Event, August 16-20, 2021, Proceedings, Part III*, volume 12827 of *Lecture Notes in Computer Science*, pages 215–244. Springer, 2021.

[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.

[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.

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

[LS15] Adeline Langlois and Damien Stehlé. Worst-case to average-case reductions for module lattices. *Des. Codes Cryptogr.*, 75(3):565–599, 2015.

[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.

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

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

A Mask refreshing

A.1 Linear mask refreshing

We recall the LinearRefresh algorithm from [RP10], working in any finite abelian group $(G, +)$:

Algorithm 12 LinearRefresh

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

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

A.2 Full mask refreshing

We recall the RefreshMasks algorithm, where the operations are performed in any group $\mathbb{G}$, for example the additive group $\mathbb{Z}_q$ for any integer q. The algorithm was proven $(n-1)$-SNI in [BBD+16].

Algorithm 13 RefreshMasks

Input: $a_1, \ldots, a_n$
Output: $c_1, \ldots, c_n$ such that $\sum_{i=1}^{n} c_i = \sum_{i=1}^{n} a_i$

1: **For** $i = 1$ **to** n **do** $c_i \leftarrow a_i$
2: **for** $i = 1$ to n **do**
3: $\quad$ **for** $j = i + 1$ to n **do**
4: $\quad\quad r \leftarrow \mathbb{G}$, $c_i \leftarrow c_i + r$, $c_j \leftarrow c_j - r$
5: $\quad$ **end for**
6: **end for**
7: **return** $c_1, \ldots, c_n$

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

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

B Proof of Theorem 2

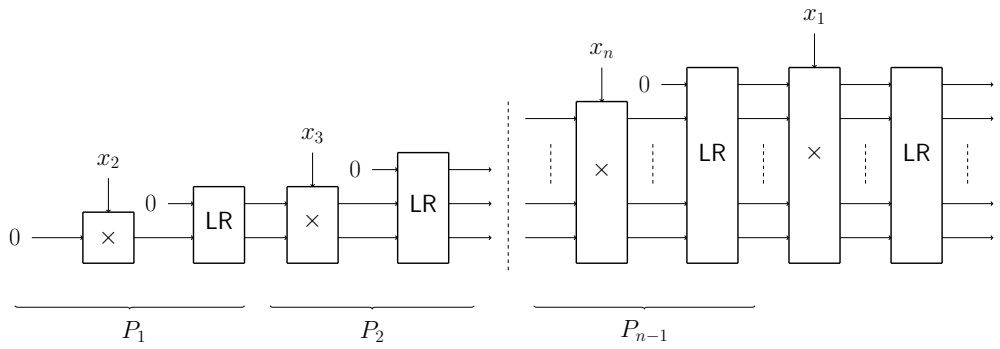

Figure 4: 1bitB2A algorithm

In order to prove the free SNI 1bitB2A we split the algorithm in several part corresponding to the process of each share. As shown in Fig. 4, we denote by P_i the core part of Alg. 2 that processes share x_{i+1}. More precisely, P_i is an algorithm that takes as input

i shares, processes x_{i+1}, and then applies a LinearRefresh on $i+1$ shares with the last input share equal to 0. Eventually, the algorithm processes x_1, and applies a LinearRefresh on the n shares, but by accumulating the randomness on the first column. The proof will focus first on proving by induction that the composition $P_{n-1} \circ \cdots \circ P_1$ achieves free SNI. Then we will prove that the last iteration, processing x_1, preserves the free SNI property.

We want to prove by induction that $P_i \circ \cdots \circ P_1$, taking as input $x_1, \ldots, x_{i+1}$ and outputting $v_1, \ldots, v_{i+1}$, satisfies the free-SNI property. This is initially true for P_1. We assume that this is the case for $P_{i-1} \circ \cdots \circ P_1$, for $i \leq n-1$. We now prove that $P_i \circ \cdots \circ P_1$ satisfies free-SNI. We use the following notations (see Fig. 5):

- $(v_j)_{1 \leq j \leq i}$: output of previous part $P_{i-1} \circ \cdots \circ P_1$,
- $(u_j)_{1 \leq j \leq i}$: result after processing x_{i+1} on v_j,
- $(r_j)_{1 \leq j \leq i}$: randomness used in LinearRefresh,
- $(w_j)_{1 \leq j \leq i+1}$: output of LinearRefresh.

We have $u_1 = (1 - 2x_{i+1})v_1 + x_{i+1}$ and $u_j = (1 - 2x_{i+1})v_j$ for $2 \leq j \leq i$. We also have $w_j = u_j + r_j \pmod q$ for $1 \leq j \leq i$. We also denote by $(w_{j,i+1})_{1 \leq j \leq i}$ the accumulated randoms on the $i+1$-th column of LinearRefresh, with $w_{j,i+1} = -(r_1 + \cdots + r_j) \pmod q$ for $1 \leq j \leq i$, with eventually the last output share $w_{i+1} = -(r_1 + \cdots + r_i) \pmod q$.

Let $\mathcal{W}$ be any set of t intermediate variables in $P_i \circ \cdots \circ P_1$. We split $\mathcal{W}$ in $\mathcal{W}_1 \cup \mathcal{W}_2$, corresponding to t_1 variables from $P_{i-1} \circ \cdots \circ P_1$ and t_2 variables from P_i, with $t = t_1 + t_2$. We can assume $t < i$. Otherwise, if $t \geq i$, then we can let $I = [2, i+1]$; in that case, we have $|I| = i \leq t$ and we can simulate all intermediate variables from inputs $x_{|I}$ since x_1 does not appear in any computation; additionally, any $O \subsetneq [1, i+1] \setminus I$ is empty and therefore the free-SNI property is satisfied.

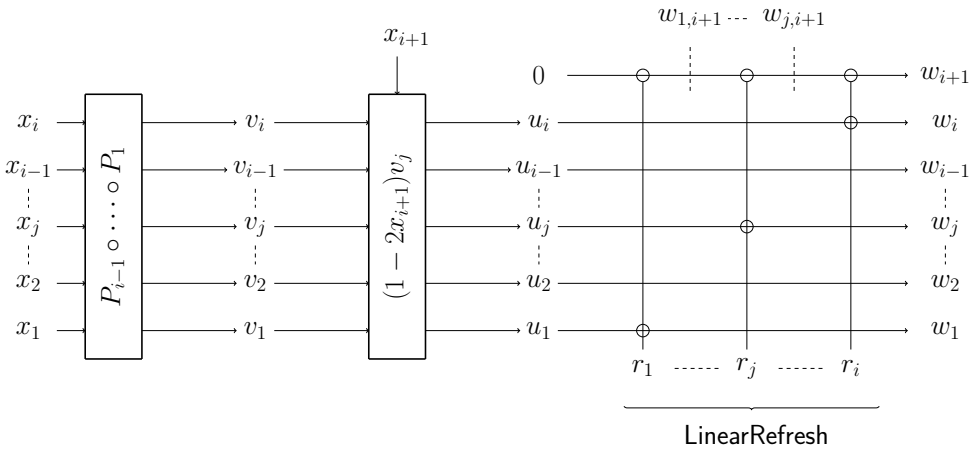

Figure 5: Core part of 1bitB2A

From the free-SNI property of $P_{i-1} \circ \cdots \circ P_1$ with t_1 probes, we can construct a set $U \subset [1, i]$, with $|U| \leq t_1$, such that the intermediate variables $\mathcal{W}_1$ and outputs $v_{|U}$ can be perfectly simulated from inputs $x_{|U}$. Additionally, for any subset $O' \subsetneq [1, i] \setminus U$, the outputs $v_{|O'}$ are uniformly and independently distributed, even conditioned on $\mathcal{W}_1$ and $v_{|U}$. We now extend the set U of input indices to a superset $I \supset U$ to allow simulation of the remaining intermediate variables $\mathcal{W}_2$ from P_i.

We initially set $I = U$. If $x_{i+1}, 1 - 2x_{i+1}$ or any $w_{j,i+1}$ belong to $\mathcal{W}_2$, we add $i+1$ to to I; note that for the variables $w_{j,i+1}$ this corresponds to a probe on the $i+1$-th column. For $1 \leq j \leq i$, if at least two variables in the j-th column belong to $\mathcal{W}_2$, namely any of

$(1-2x_{i+1})v_j, v_j + x_{i+1}, u_j + r_j$ or r_j, we add both $i+1$ and j to I. If a single variable in the j-th column has been probed, we add j to I if $j \notin U$, otherwise we add $i+1$ to I. By construction we must have $|I| \leq |U| + t_2 \leq t_1 + t_2 = t$. We consider any $O \subsetneq [1, i+1] \setminus I$, and we now show that:

1. We can perfectly simulate $\mathcal{W}$ and outputs $w_{|I}$ from inputs $x_{|I}$,
2. The outputs $w_{|O}$ are uniformly and independently distributed, even conditioned on $\mathcal{W}$ and $w_{|I}$.

For this, we distinguish 2 cases, depending on whether $i+1 \in I$ or $i+1 \notin I$:

$\underline{i+1 \in I.}$ Since we know x_{i+1}, the t_2 variables of $\mathcal{W}_2$ and outputs $w_{|I}$ can be perfectly simulated from $v_{|I\setminus\{i+1\}}$ and x_{i+1}, by propagating the simulation column by column. From the free-SNI of $P_{i-1} \circ \cdots \circ P_1$, we know outputs $v_{|U}$ can be perfectly simulated from $x_{|U}$, and therefore from $x_{|I}$. It remains to show how to simulate the outputs $v_{|I\setminus(U\cup\{i+1\})}$, and to prove that the output variables $w_{|O}$ of $P_i \circ \cdots \circ P_1$ are uniformly distributed.

For this, we construct a subset $O' \subsetneq [1, i] \setminus U$ of indices on which we apply induction hypothesis, namely the free-SNI of $P_{i-1} \circ \cdots \circ P_1$, such that the outputs $v_{|O'}$ are uniformly distributed. We let:

$$O' = O \cup I \setminus (U \cup \{i+1\})$$

To apply the induction hypothesis, we must verify that $O' \subsetneq [1, i] \setminus U$. We have $O \subset [1, i] \setminus I \subset [1, i] \setminus U$ and $I \setminus (U \cup \{i+1\}) \subset [1, i+1] \setminus \{i+1\} \setminus U \subset [1, i] \setminus U$, which gives $O' \subset [1, i] \setminus U$. To show the inclusion is strict, we show that $[1, i] \setminus U \setminus O'$ is non-empty:

$$\begin{aligned}
([1, i] \setminus U) \setminus O' &= [1, i] \setminus (U \cup O') \\
&= [1, i] \setminus (U \cup O \cup (I \setminus (U \cup \{i+1\}))) \\
&= [1, i] \setminus (O \cup (I \setminus \{i+1\})) && U \subset I \\
&= ([1, i] \setminus (I \setminus \{i+1\})) \setminus O \\
&= [1, i+1] \setminus I \setminus O && i+1 \in I
\end{aligned}$$

Since by assumption $O \subsetneq [1, i+1] \setminus I$, the set $[1, i+1] \setminus I \setminus O$ is non-empty and so is $[1, i] \setminus U \setminus O'$. Hence from the free-SNI of $P_{i-1} \circ \cdots \circ P_1$, the outputs $v_{|O'}$ are uniformly and independently distributed. Therefore, we can perfectly simulate $v_{|I\setminus(U\cup\{i+1\})}$ with fresh randoms, and therefore all variables $v_{|I\setminus\{i+1\}}$ can be simulated. As explained above, knowing x_{i+1}, the simulation can then be propagated to P_i, and finally the intermediate variables $\mathcal{W}$ and outputs $w_{|I}$ can be perfectly simulated from inputs $x_{|I}$.

We now show that the outputs $w_{|O}$ are uniformly and independently distributed, where w_j is the result of $u_j + r_j$ for $j \in O$, for $1 \leq j \leq i$; note that since $i+1 \in I$, we must have $i+1 \notin O$. We already know that the outputs $v_{|O}$ are uniformly and independently distributed and we remark that for $j \in O$, u_j is a bijective transformation of v_j. Namely $u_j = (1-2x_{i+1})v_j$ for $j \geq 1$ and $u_1 = (1-2x_{i+1})v_1 + x_{i+1}$. Therefore $u_{|O}$ are uniformly and independently distributed. Eventually u_j acts as a one-time pad on $w_j = u_j + r_j$ and we conclude that the variables $w_{|O}$ are uniformly distributed, even conditioned on $w_{|I}$ and $\mathcal{W}$.

$\underline{i+1 \notin I.}$ We first simulate outputs $w_{|I}$ and intermediate variables $\mathcal{W}$. We consider $j \in I$. If $j \in U$, then by construction none of $(1-2x_{i+1})v_j, u_j, r_j, w_j, w_{j,i+1}$ belongs to $\mathcal{W}_2$, otherwise we would have $i+1 \in I$. In particular, since r_j is a fresh uniform random and not involved in the computation of any intermediate variable from $\mathcal{W}$, it acts as a one-time pad on the value $u_j + r_j$. Therefore all outputs $w_{|U}$ can be perfectly simulated uniformly and independently.

We now simulate outputs $w_{|I\setminus U}$. For this, we let $O' = I \setminus U$. We have $O' \subsetneq [1, i] \setminus U$. Indeed, we have assumed $t < i$, and from $|I| \leq t < i$, we have $I \subsetneq [1, i]$; from $U \subset I$, we deduce $O' = I \setminus U \subsetneq [1, i] \setminus U$. From the induction hypothesis, we know that $v_{|O'} = v_{|I\setminus U}$ are uniformly and independently distributed. Since the application $v \mapsto (1 - x_{i+1})v$ is

a bijection, then $(1-2x_{i+1})v_j$ is uniformly distributed. Moreover, for $j \in I \setminus U$, then by construction of I only one of $(1-2x_{i+1})v_j, r_j, u_j + r_j$ belongs to $\mathcal{W}_2$. Hence, we can perfectly simulate $(1-2x_{i+1})v_j$ with uniform random without knowledge of x_{i+1}, since the distribution of v_j is independent of $\mathcal{W} \setminus \{(1-2x_{i+1})v_j\}$. Simulation of any of r_j or $u_j + r_j$ can also be made with fresh uniform random since r_j acts as a one-time pad. Eventually, once the variable from $\mathcal{W}$ in the j-th column has been simulated, the simulation can be propagated in the rest of the column and eventually to w_j. Therefore, we can perfectly simulate variables $\mathcal{W}$ and outputs $w_{|I}$ from inputs $x_{|U} \subset x_{|I}$.

It remains to show that the outputs $w_{|O}$ are uniformly and independently distributed. For $1 \leq j \leq i$, $w_j = u_j + r_j \pmod q$, where r_j acts as a one-time pad (neither r_j nor $w_{j-1,i+1} - r_j$ belongs to $\mathcal{W}$). If $i+1 \in O$, since $O \subsetneq [1, i+1] \setminus I$, there exists $j^\star \in [1, i+1] \setminus (I \cup O)$. We have that $r_{j^\star}$ a uniform random independent from any of intermediate variable $\mathcal{W}$ and outputs $w_{|I \cup (O \setminus \{i+1\})}$. Therefore the output $w_{i+1} = -\sum_{j \neq j^\star} r_j - r_{j^\star}$ is uniformly distributed, even conditioned on $\mathcal{W} \cup w_{|I \cup (O \setminus \{i+1\})}$. Eventually, the outputs $w_{|O}$ are uniformly and independently distributed.

We conclude that $P_i \circ \cdots \circ P_1$ is free-SNI, which achieves the induction. Therefore we have that $P_{n-1} \circ \cdots \circ P_1$ is free-SNI.

It remains to show that the last part that processes x_1 preserves the free-SNI property. As previously, let $\mathcal{W}$ be any set of t intermediate variables, we split $\mathcal{W} = \mathcal{W}_1 \cup \mathcal{W}_2$ corresponding to variables from $P_{n-1} \circ \cdots \circ P_1$ and the last part respectively, where $|\mathcal{W}_1| = t_1$ and $|\mathcal{W}_2| = t_2$. If $t_2 = 0$, then there is nothing left to prove since $P_{n-1} \circ \cdots \circ P_1$ is free$-$SNI and any set of $\leq n-1$ outputs is uniformly and independently distributed from the LinearRefreshMasks procedure. Hence, we assume $t_2 > 0$. Additionally, we assume $t < n$. Otherwise, we can take $I = [1, n]$ and we can perfectly simulate all variables since we provide all inputs to the simulator.

The proof is very similar to the previous part. First, from the free-SNI of $P_{n-1} \circ \cdots \circ P_1$, we deduce that there exists $U \subset [1, n]$, with $|U| \leq t_1$, such that the intermediate variables $\mathcal{W}_1$ and $P_{n-1} \circ \cdots \circ P_1$ outputs $v_{|U}$ can be perfectly simulated from inputs $x_{|U}$. Moreover for any subset $O \subsetneq [1, n] \setminus U$, outputs $v_{|O}$ are uniformly and independently distributed.

We extend the set U as previously. Initially, we set $I = U$. As previously, we denote by $(w_{j,1})_{1 \leq j \leq i}$ the accumulated randoms on the first column of the last LinearRefresh. If any of x_1, $1 - 2x_1$, $w_{j,1}$ belongs to $\mathcal{W}_2$, we add 1 to I. Otherwise, for $2 \leq j \leq n$, if at least two of v_j, r_j, w_j belong to $\mathcal{W}_2$, we add both 1 and j to I. If only one of v_j, r_j, w_j belongs to $\mathcal{W}_2$, we add j to I if $j \notin U$ or 1 to I if $j \in U$. By construction, $|I| \leq |U| + t_2 \leq t_1 + t_2 = t$.

Let $O \subsetneq [1, n] \setminus I$. We now show how to simulate variables $\mathcal{W}$ and outputs $w_{|I}$ from inputs $x_{|I}$, and we show that outputs $w_{|O}$ are uniformly and independently distributed. We distinguish two cases whether $1 \in I$ or $1 \notin I$:

$\underline{1 \in I.}$ By construction, any intermediate variable from $\mathcal{W}_2$ and outputs $w_{|I}$ can be perfectly simulated from $v_{|I}$, using the knowledge of x_1. We let $O' := O \cup (I \setminus U)$. Using $O \subsetneq [1, n] \setminus I$, and since $[1, n] \setminus I$ and $I \setminus U$ have empty intersection, we deduce $O' = O \cup (I \setminus U) \subsetneq ([1, n] \setminus I) \cup (I \setminus U) = [1, n] \setminus U$. Therefore, from the free-SNI of $P_{n-1} \circ \cdots \circ P_1$, the outputs $v_{|O'}$ are uniformly and independently distributed. Hence, $v_{|I \setminus U}$ can be perfectly simulated with fresh randoms. Additionally, $v_{|U}$ and variables $\mathcal{W}_1$ can be perfectly simulated from inputs $x_{|U}$. Therefore, by propagating the simulation, we can perfectly simulate intermediate variables $\mathcal{W}$ and outputs $w_{|I}$ from inputs $x_{|U} \subset x_{|I}$. Eventually, following the same argument as above, the outputs $w_{|O} = u_{|O} + r_{|O}$ are uniformly distributed, following the fact that $u_{|O}$ are bijective transformation of the uniformly distributed $v_{|O}$.

$\underline{1 \notin I.}$ We first simulate variables $\mathcal{W}$ and outputs $w_{|I}$. For $j \in U$, by construction none of the variables $(1-2x_1)v_j, u_j, r_j, w_j, w_{j,1}$ belongs to $\mathcal{W}_2$, since otherwise we would have $1 \in I$. In particular, r_j is a uniform random independent from $\mathcal{W}$ and acts as a one time pad on $w_j = u_j + r_j \pmod q$. Hence, the output w_j can be perfectly simulated with uniform random.

As previously, to simulate the output variables $w_{|I\setminus U}$, we let $O' := I \setminus U \subsetneq [1,n] \setminus U$, as we assumed $t < n$. From the free-SNI of $P_{n-1} \circ \cdots \circ P_1$, we have that the outputs $v_{|O'}$ are uniformly distributed. For $j \in I \setminus U$, then only one of $(1-2x_1)v_j, r_j, u_j + r_j$ belongs to $\mathcal{W}_2$. As previously, we can simulate any intermediate variables in column $j \in I \setminus U$ with fresh uniform random and propagate the simulation to the output w_j. Eventually, we conclude that intermediate variables and outputs $w_{|I}$ can be perfectly simulated from inputs $x_{|U} \subset x_{|I}$.

It remains to show that outputs $w_{|O}$ have the uniform distribution. This holds because for $j \in O$ and $2 \leq j \leq n$, we have $w_j = u_j + r_j \pmod q$, where r_j is a fresh uniform random independent from $\mathcal{W}$ and therefore acts as a one time pad. Finally, if $1 \in O$, there exists $j^\star \in [1,n] \setminus (O \cup I)$, and we can write $w_1 = (v_1 - \sum_{j \neq j^\star} r_j) - r_{j^\star} \pmod q$, where $r_{j^\star}$ is a fresh uniform random independent from $\mathcal{W}$ and other outputs; therefore it acts as a one-time-pad for w_1.

Eventually, we have shown that the whole algorithm achieves free-SNI security.

C Nonce generation with approximate Boolean to arithmetic conversion

In Section 4.2, we have described an approximate Boolean to arithmetic conversion algorithm such that

$$y_1 + \cdots + y_n = x + e \pmod q$$

for some centered error $|e| \leq e_{max}$, with $e_{max} = \lfloor n/2 \rfloor$. We can ensure that x is uniformly distributed in the interval $]-\gamma_1, \gamma_1]$, where γ_1 is a power-of-two. Therefore, the distribution of $x' = x + e$, conditioned on $x' \in]-\gamma_1 + e_{max}, \gamma_1 - e_{max}]$, is uniform. This implies that the faster approximate Boolean to arithmetic conversion can also be used in the signature generation of Dilithium, by using a slightly stricter rejection sampling.

More precisely, instead of performing the rejection sampling on $\mathbf{z} = \mathbf{y} + c\mathbf{s}_1$ with $\|\mathbf{z}\|_\infty \geq \gamma_1 - \beta$, we perform the slightly stricter rejection sampling $\|\mathbf{z}\|_\infty \geq \gamma_1 - \beta - e_{max}$. We show below that the conditional probability distribution of each coefficient of $\mathbf{z}$ remains uniform in an interval, which implies that no information is leaked about the secret-key. We also show that this induces only a very small increase in the number of repetitions to generate a valid signature.

Analysis without error. We start with no error ($e = 0$), following the reasoning in [BDK+21, Section 3.4]. The probability that $\|\mathbf{z}\|_\infty < \gamma_1 - \beta$ is computed by considering each coefficient separately. For each coefficient σ of $c\mathbf{s}_1$, the corresponding coefficient $z = y + \sigma$ of $\mathbf{z} = \mathbf{y} + c\mathbf{s}_1$ will be in the interval $[-\gamma_1 + \beta + 1, \gamma_1 - \beta - 1]$ whenever the corresponding coefficient y of $\mathbf{y}$ is in $[-\gamma_1 + \beta + 1 - \sigma, \gamma_1 - \beta - 1 - \sigma]$. Since y is uniformly distributed in $[-\gamma_1 + 1, \gamma_1]$ and $|\sigma| \leq \beta$, this happens with probability $(2(\gamma_1 - \beta) - 1)/(2\gamma_1 - 1)$, and moreover the conditional distribution of z is uniform in $[-\gamma_1 + \beta + 1, \gamma_1 - \beta - 1]$, as required. Therefore the probability that every coefficient of $\mathbf{y}$ is in the good range is:

$$\left(\frac{2(\gamma_1 - \beta) - 1}{2\gamma_1 - 1}\right)^{256 \cdot \ell} = \left(1 - \frac{\beta}{\gamma_1 - 1/2}\right)^{256 \cdot \ell} \simeq \exp(-256 \cdot \beta\ell/\gamma_1)$$

Similarly, the authors provide a (heuristic) estimate that $\|\mathbf{r}_0\|_\infty < \gamma_2 - \beta$, and obtain a probability $\exp(-256 \cdot \beta k/\gamma_2)$, so that the probability that the two conditions are satisfied is $\simeq \exp(-256 \cdot \beta(\ell/\gamma_1 + k/\gamma_2))$ where (k, ℓ) are the dimensions of $\mathbf{A}$.

Analysis with centered error. We now consider the case of a centered error $|e| \leq e_{max}$, with $e_{max} = \lfloor n/2 \rfloor$, where we obtain for each coefficient $z = y + \sigma + e$, instead of $z = y + \sigma$. We now consider the stricter condition $\|\mathbf{z}\|_\infty < \gamma_1 - \beta'$ with $\beta' = \beta + e_{max}$. As previously, a coefficient $z = y + \sigma + e$ will be in the interval $[-\gamma_1 + \beta' + 1, \gamma_1 - \beta' - 1]$ whenever y is in $[-\gamma_1 + \beta' + 1 - \sigma - e, \gamma_1 - \beta' - 1 - \sigma - e]$. Since y is uniformly distributed in $[-\gamma_1 + 1, \gamma_1]$ and $|\sigma + e| \leq \beta'$, this happens with probability $(2(\gamma_1 - \beta') - 1)/(2\gamma_1 - 1)$, and moreover the conditional distribution of z is still uniform in $[-\gamma_1 + \beta' + 1, \gamma_1 - \beta' - 1]$, as required. As previously, the probability that every coefficient of $\mathbf{y}$ is in the good range is $\simeq \exp(-256 \cdot \beta' \ell/\gamma_1) = \exp(-256 \cdot (\beta + e_{max})\ell/\gamma_1)$ and therefore the probability that the two conditions are satisfied is $\simeq \exp\left(-256 \cdot ((\beta + e_{max})\ell/\gamma_1 + \beta k/\gamma_2)\right)$ where $e_{max} = \lfloor n/2 \rfloor$. We obtain the following number of repetitions, which show only a very small increase with the security order.

Table 9: Number of repetitions for security level 2, with $n = t + 1$ shares

Security order t	0	2	3	4	5	6	8	10	12
Repetitions	4.25	4.29	4.32	4.32	4.36	4.36	4.39	4.42	4.46

D Randomness consumption

Table 10: Randomness usage for the gadgets in number of calls to `rand32()`.

	Security order t					
	1	2	3	4	5	6
CGV14	1	99	198	424	684	975
Alg. 3 with ShiftMod	64	160	288	448	640	864
DecomposeComp	52	154	306	508	760	1062
DecomposeMod	78	284	568	1030	1570	2238
[SPOG19]	18	54	108	180	270	378
BtoAqApprox	2	7	18	41	88	183
BtoAqExact	6	19	42	81	148	267
B2A mod 2^k [BCZ18]	2	7	18	41	88	183