Masking Dilithium

Efficient Implementation and Side-Channel Evaluation

Vincent Migliore[1], Benoît Gérard[2,3], Mehdi Tibouchi[4], and Pierre-Alain Fouque[2]

[1] LAAS–CNRS, Univ. Toulouse, CNRS, INSA
vincent.migliore@laas.fr
[2] Univ Rennes, CNRS, IRISA
{benoit.gerard,pierre-alain.fouque}@irisa.fr
[3] Direction Générale de l'Armement
[4] NTT Corporation
mehdi.tibouchi.br@hco.ntt.co.jp

Abstract. Although security against side-channel attacks is not an explicit design criterion of the NIST post-quantum standardization effort, it is certainly a major concern for schemes that are meant for real-world deployment. In view of the numerous physical attacks that have been proposed against post-quantum schemes in recent literature, it is in particular very important to evaluate the cost and effectiveness of side-channel countermeasures in that setting.

For lattice-based signatures, this work was initiated by Barthe et al., who showed at EUROCRYPT 2018 how to apply arbitrary order masking to the GLP signature scheme presented at CHES 2012 by Güneysu, Lyubashevsky and Pöppelman. However, although Barthe et al.'s paper provides detailed proofs of security in the probing model of Ishai, Sahai and Wagner, it does not include practical side-channel evaluations, and its proof-of-concept implementation has limited efficiency. Moreover, the GLP scheme has historical significance but is not a NIST candidate, nor is it being considered for concrete deployment.

In this paper, we look instead at Dilithium, one of the most promising NIST candidates for postquantum signatures. This scheme, presented at CHES 2018 by Ducas et al. and based on module lattices, can be seen as an updated variant of both GLP and its more efficient sibling BLISS; it comes with an implementation that is both efficient and constant-time.

Our analysis of Dilithium from a side-channel perspective is threefold. We first evaluate the side-channel resistance of an ARM Cortex-M3 implementation of Dilithium without masking, and identify exploitable side-channel leakage. We then describe how to securely mask the scheme, and verify that the masked implementation no longer leaks. Finally, we show how a simple tweak to Dilithium (namely, replacing the prime modulus by a power of two) makes it possible to obtain a considerably more efficient masked scheme, by a factor of 7.3 to 9 for the most time-consuming masking operations, without affecting security.

1 Introduction

Post-quantum cryptography and lattice-based signatures. As the threat of quantum computers becomes increasingly concrete, the need for public-key cryptography to transition away from legacy schemes based on factoring and discrete logarithms and towards post-quantum secure primitives gets more pressing. In particular, there is a growing push to make post-quantum cryptography, which was of somewhat theoretical interest for some time, ready for real-world deployment. At the forefront of that push is NIST's post-quantum standardization process [1], which aims at selecting post-quantum secure schemes for encryption and signatures that can practically replace RSA and elliptic curve cryptography. The first round includes 69 candidates across encryption and signatures, based on codes, lattices, multivariate cryptography, hash functions and more.

Among them, lattice-based schemes stand out as particularly attractive, thanks to their strong security foundations and their high level of efficiency, often comparable to RSA and elliptic curves both in terms of key and ciphertext/signature size, and of computational complexity. However, they present a unique set of challenges from an implementation perspective, due to the reliance on new types of operations such as Gaussian sampling, polynomial arithmetic, number-theoretic transforms and rejection sampling.

Such new operations are a concern, in particular, from the standpoint of fault and side-channel analysis. A number of implementation attacks have been proposed against lattice-based schemes, including fault attacks [4,11], cold boot attacks [2], cache timing attacks [13,17] and more standard power/electromagnetic analysis [12], taking advantage of vulnerabilities of the implementation of those new operations in order to mount key recovery attacks. Lattice-based signatures have notably been the target of multiple such attacks. It is therefore of prime importance to study how to securely and efficiently protect implementations against those attacks.

Masking lattice-based signatures. Regarding side-channels, a generic and provable countermeasure is known: masking, in which all sensitive variables in the signing algorithm is stored and processed as several shares, typically using some linear secret sharing scheme. The two most common approaches are *boolean masking*, where a secret bitstring x is represented as the bitwise XOR $x = x_1 \oplus \cdots \oplus x_t$ of uniformly random shares x_i's, and *arithmetic masking*, where a secret element x of $\mathbb{Z}/m\mathbb{Z}$ is represented as the sum $x = x_1 + \cdots + x_t$ modulo m of uniformly random elements of $\mathbb{Z}/m\mathbb{Z}$. Boolean masking is better suited to mask logical operations, whereas arithmetic masking is convenient for operations than can be represented in a simple way as arithmetic circuits (i.e., multivariate polynomials modulo m).

Applying masking countermeasures to lattice-based signatures is a challenging task, mainly due to the overall structure of the corresponding signing algorithm, which typically involve sampling some sensitive randomness, combining it with the secret key, and then carrying out some form of rejection sampling on the resulting value. The random sampling and rejection sampling are complicated operations which are better suited for boolean masking, whereas the main part of the signing algorithm involving the secret key is linear modulo some prime p, and therefore convenient for arithmetic masking. Protecting the entire algorithm therefore requires conversions between arithmetic and boolean masking, targeted unmasking of provably non-sensitive variables, and the design of novel masked gadgets to support the new sampling and rejection operations.

This was all first tackled recently by Barthe et al. [3] in a EUROCRYPT 2018 paper providing a complete, arbitrary order masking of the (relatively simple) lattice-based signature scheme of Güneysu, Lyubashevsky and Pöppelman (GLP). The paper addresses all the issues above in the case of GLP to construct a provably secure masked implementation of the key generation and signing algorithms of GLP. It suffers from several limitations, however. First, the GLP scheme itself has the advantage of being relatively simple compared to later lattice-based signatures like BLISS and the current NIST candidates, but it is of limited practical relevance, due to a level of efficiency that falls short of the state of the art, and more lax security guarantees. Second, the masked implementation of Barthe et al. incurs a rather severe overhead compared to the (already not that efficient) unmasked scheme. And finally, although the paper comes with security proofs, it does not include a practical side-channel evaluation: this can be a problem in practice due to discrepancies between formal specifications and compiled code, unexpected data dependencies introduced at the CPU-level, and other hardware issues like glitches.

Our contributions. As a result, it is desirable to consider the application of the masking countermeasure to a more up-to-date lattice-based signature scheme (preferably a NIST candidate), hopefully achieving better performance than the masked implementation of Barthe et al., and with a concrete validation of side-channel resistance.

This is the goal pursued in this work, where we examine in particular the Dilithium signature scheme of Ducas et al. [10], a NIST candidate that can be seen as a descendant of both GLP and BLISS. It comes with an implementation that emphasizes both efficiency and constant running time (so as to achieve security against timing attacks and simple power analysis). In particular, like GLP but unlike BLISS, its main variant excludes Gaussian distribution and only relies on random numbers that are sampled uniformly from small intervals.

Our main contributions are as follows:

1. we carry out a side-channel evaluation of the reference design of Dilithium when implemented on an ARM Cortex-M3 micro-controller (the STM32F1), and identify exploitable side-channel leakage, which underscores the need for suitable countermeasures;
2. we propose an efficient masking of Dilithium at any order, partially leveraging the work carried out by Barthe et al. on GLP (in particular, we reuse their formally verified masked gadgets);

3. we describe a simple variant of Dilithium that lends itself to a considerably more efficient masking while preserving security, using the key idea of switching from a prime modulus to a power of two[5];
4. we implement these masked schemes on the same ARM Cortex-M3 micro-controller, we manage to remove unexpected leakages due to some micro-architectural features and evaluate both the efficiency and side-channel resistance of the implementation, with satisfactory results on both counts.

The paper is organized as follows. Section 2 recalls the key generation and the signing algorithms of Dilithium. Section 3 evaluates the side-channel leakage of sensitive operations on our STM32F1 target micro-controller. Section 4 proposes an efficient masking of the Dilithium reference design, as well as that of our proposed variant (using a power-of-two modulus) which greatly improves masking efficiency. Section 5 provides implementation results, both in terms of performance and of side-channel resistance.

2 The Dilithium Signature scheme

Dilithium is a signature scheme based on Lyubashevsky's Fiat–Shamir with aborts framework and is based on hard problems in module lattices. Its core functions are *KeyGen* for the key generation, *Sign* to produce a signature of a message, and *Verify* to verify the signature.

One of the main features of Dilithium (aside from its module lattice approach) is the key compression mechanism to reduce public key size. The compression is performed at two different levels. First, Module matrices are constructed with an extendable output function (XOF), which generates a (deterministic) pseudo-random string from a small seed. Thus, the public only requires the seed and not the full matrix. Second, the public key size is reduced using a truncation on its second component. This truncation is performed coefficient-wise and is associated to an error-correcting code mechanism to recover truncated bits[6].

In addition, Dilithium does not instantiate Module with discrete Gaussian sampling, but with bounded coefficients. This approach greatly simplifies the arithmetic of Dilithium (and at the same time masking) since discrete Gaussian sampling is much more complex than a simple bound check.

In this paper, we mainly focus on the key generation and the signature generation algorithms (which will respectively be called DILITHIUM.KeyGen and DILITHIUM.Sign) since the verification algorithm does not handle sensitive data and hence does not require masking.

DILITHIUM.KeyGen. The DILITHIUM.KeyGen algorithm, described in Algorithm 1, generates the secret key S_{key} and public key P_{key} required to respectively sign and verify a message.

[5] this statement is discussed later on in Section 4.4.

[6] For a formal description of the different truncation procedures used in Dilithium (namely Decompose_q, HighBits_q, LowBits_q and Power2Round) the reader can refer to the original Dilithium paper [9].

Algorithm 1 DILITHIUM.KeyGen()

```
1: $\rho, \rho' \leftarrow \{0,1\}^{256}$
2: $A = \mathrm{Sam}(\rho) \in R_q^{k\times\ell}$
3: $(S_1, S_2) = \mathrm{Sam}(\rho') \in R_\eta^{\ell\times 1} \times R_\eta^{k\times 1}$
4: $T = A \cdot S_1 + S_2 \in R_q^{k\times 1}$
5: $T_1 = \mathrm{Power2Round}(T, d) \in R_q^{k\times 1}$
6: $P_{key} = (\rho, T_1)$
7: $S_{key} = (\rho', S_1, S_2, T)$
8: return $(P_{key}, S_{key})$
```

The randomness is obtained using an extendable output function (XOF) called Sam which takes a random seed as input and returns an extendable pseudo-random string. The Sam function is used to compute the matrix A (which is part of the public key) and matrices (S_1,S_2) (which are part of the secret key). Unlike coefficients of A, the coefficients of S_1 and S_2 are *small* ones.

Regarding arithmetic complexity, the Sam function and the polynomial multiplication line 4 are the most time-consuming part of the computation. For the implementation provided for the NIST competition, the Sam function is implemented using SHAKE-256, and polynomial multiplications with NTT algorithm.

DILITHIUM.Sign. The DILITHIUM.Sign algorithm is described in Algorithm 2. It is constructed by a rejection sampling loop where a fresh signature is generated until it satisfies some security properties. First of all, a uniformly sampled matrix Y in R_{γ_1-1} is secretly generated, and multiplied by the public value A to produce W (lines 6 and 7). Then a challenge $C \in B_{60}$ is generated as the output of a hash function H with (ρ, T_1, W_1, μ) as input, where W_1 is composed by the high order bits of W and μ is the message to sign.

To ensure that the signature does not leak information about the key, line 11 executes some bound checks. If this verification fails, a new signature is generated. One of the most important parameter is β, because it will determine the number of rounds required before a valid signature is produced. For recommended parameters, an average of 5 rounds are needed before producing a good set of parameters. Eventually, the $\mathrm{MakeHint}_{q,2\gamma_2}$ procedure line 12 will generate some hints for the public key reconstruction (bits are due to its truncation.).

3 Side-channel evaluation of unmasked Dilithium

In this section we report the results we obtained evaluating the potential side-channel weaknesses of an unprotected implementation of Dilithium. We performed Welch's t-test to localize potential leakages and single-bit DPA on secret variables to confirm that actually correspond to exploitable leakages.

Algorithm 2 DILITHIUM.Sign(S_{key}, μ)

```
1: $A = \mathrm{Sam}(\rho) \in R_q^{k\times\ell}$
2: $T_1 = \mathrm{Power2Round}(T, d) \in R_q^{k\times 1}$
3: $T_0 = T - T_1 \cdot 2^d \in R_q^{k\times 1}$
Rejection sampling loop
    4: $\rho'' \leftarrow \{0,1\}^{256}$
    5: $Y = \mathrm{Sam}(\rho'') \in R_{\gamma_1-1}^{\ell\times 1}$
    6: $W = A \cdot Y \in R_q^{k\times 1}$
    7: $W_1 = \mathrm{HighBits}_{q,2\gamma_2}(W) \in R_q^{k\times 1}$
    8: $C = \mathrm{H}(\rho, T_1, W_1, \mu) \in \{0,1\}^{256}$
    9: $Z = Y + CS_1 \in R_q^{\ell\times 1}$
    10: $R_0 = \mathrm{LowBits}_{q,2\gamma_2}(W - CS_2)$
    11: if $||Z||_\infty \geq \gamma_1 - \beta$ or $||R_0||_\infty \geq \gamma_2 - \beta$ or $||CT_0||_\infty \geq \gamma_2$ goto 4
12: $H = \mathrm{MakeHint}_{q,2\gamma_2}(-CT_0, W - CS_2 + CT_0)$
13: return $(Z, H, C)$
```

Operation choice motivation. We limited the unprotected-case study to three operations namely, the rejection, $\mathrm{LowBits}_{q,2\gamma_2}$ and $\mathrm{HighBits}_{q,2\gamma_2}$. We detail now the motivations that led to this choice.

The rejection is one of the most critical operations as it is both used for secret data generation and for rejection sampling during the signature computation. A successful attack on the rejection will leak information on S_1, S_2 during the key generation, on Y during the signature or on a rejected Z (which leaks information about S_1 as stated by the designers). Regarding decomposition operations, $\mathrm{LowBits}_{q,2\gamma_2}(W - C \cdot S_2)$ in line 10 of Algorithm 2 and $\mathrm{HighBits}_{q,2\gamma_2}$ which is part of the computation of $\mathrm{MakeHint}_{q,2\gamma_2}(-CT_0, W - CS_2 + CT_0)$ (line 12) have been chosen because $W - C \cdot S_2$ is a sensitive variable since, together with the public value, Z it would allow the attacker to recover the secret key T.

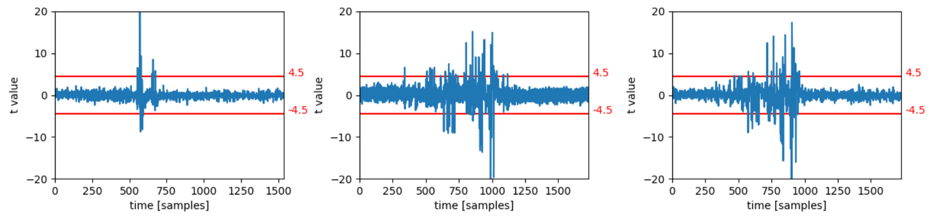
We did not studied the Sam function. Although it is a good candidate for an attack as it is used to generate S_1, S_2 and Y, its actual implementation can vary from a Dilithium implementation to another. Indeed, designers of Dilithium state that *different implementations are free to use whichever pseudo-random generator is offering the best performance and security on their respective platform.* The situation is similar for the random oracle H as its actual implementation from the NIST submission relies on SHAKE256 what is not mandatory. Studying the resistance of these primitives is indeed of great importance before deploying a solution but is out of the scope of this paper where we aim at considering intrinsic security properties of DILITHIUM.

Note that the polynomial multiplications used to compute $T = A \cdot S_1 + S_2$ during the key generation (line 4 of Algorithm 1) and $W = A \cdot Y$ during the signature is also a sensitive step of the algorithm. Since this classical operation has already been shown to be sensitive to side-channel attacks and is easy to mask (due to its linearity) we did not evaluate its unprotected version.

Experimental setup and methodology. Our workbench were composed of an STM32F1 micro-controller from a discovery platform (referred as the DUT in the rest of the section) running sensitive operations, an H 2.5-2 near-field probe coupled with a 20dB pre-amplifier to measure electromagnetic leaks, an instrumented RTO2014 oscilloscope from Rohde & Schwarz (with 1GHz bandwidth) to capture traces and a desktop computer for performing trace analysis.

The oscilloscope was configured with a sample rate ensuring 8 samples per DUT clock cycle (that is a bit more than 160 MHz). The data was sent to the DUT through a serial connection, then before the computation a trigger helped the synchronization of the oscilloscope and the DUT (using a GPIO pin of the board). A python script was used to perform t-test and DPA on the captured traces. For the t-test we used the fixed vs random approach and took care of randomly mix requests from both populations. The single-bit DPA has been performed on each bit of the sensitive data in the input of the target operations.

Evaluation results. We present here the results obtained. For the t-test (Figure 1), the threshold use is the classical 4.5 one (red lines).



Rejection $\quad$ $\mathrm{LowBits}_{q,2\gamma_2}$ $\quad$ $\mathrm{HighBits}_{q,2\gamma_2}$

Fig. 1: T-Test evaluation for targeted operations (using 500 traces).

As can be seen in Figure 1, basic implementation are highly leaking (we observe clear peaks using only 500 traces). In all cases, we confirmed the threat induced by those leakages by computing single-bit DPA curves for all sensitive inputs. Results can be seen in Figure 2 and show that t-test peaks are actual leakages. We obtain similar results for other target bits even if for some bits the signal has a smaller magnitude.

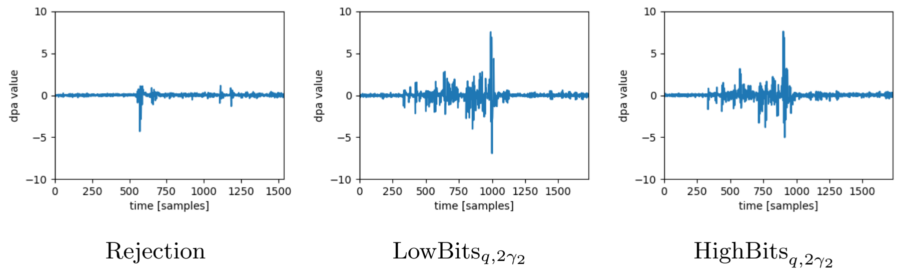


Fig. 2: Single-bit DPA curves on bit 0 of sensitive data (using 5000 traces).

4 Masking Dilithium

Results of Section 3 confirm that an attacker having a physical access to a device can easily perform a side-channel key-recovery on a standard Dilithium implementation. In this section, we propose some guidelines to efficiently protect the Dilithium algorithm.

First, we provide some information about the leakage model adopted for the determination of masking operations. Second, we present a high-level strategy for masking. Third, we detail the implementation of secured operations.

4.1 Leakage model

The first introduced side-channel security model was the noisy leakage model in which the attacker obtains sensitive information mixed with noise [5,18]. The main limitation of this approach is the deep knowledge of the noise it requires which is strongly device-dependent.

A more generic approach is the probing model [14]. In the t-probing model, the attacker observes t intermediate noise-free variables of the algorithm (as if she was directly probing the bus). In [8], a reduction have been obtained proving that security in the t-probing model implies security in the noisy leakage one.

This last model is the one to consider in the case a designer wants to totally remove leakages up to a given order. To achieve probing security, operations on secret variables are computed over shared values, i.e. variables which are split into shares containing partial information of the initial variable mixed with noise. Masking variables at order d requires at least $d+1$ shares. The threshold probing model introduces the notion of *t-probing secure gadget.*

Definition 1 *A circuit G is a t-probing secure gadget if and only if every tuple composed of t of its intermediate variables is independent from any sensitive variables it manipulates.*

In the following, we expose our masking strategy and describe the secure gadgets used for our implementation.

4.2 Presentation of the masked key generation and signature

We provide here design considerations on securing DILITHIUM.Keygen and DILITHIUM.Sign in the t-probing model. The sensitive operations performed are of different natures which implies using both arithmetic and Boolean masking. In the following, we help the reader by disambiguating the used masking using the prefixes `arith::` for arithmetic (the sensitive variable is the sum of the shares) and `bool::` for Boolean masked operations (the sensitive variable is the exclusive or of the shares).

Masking of DILITHIUM.Keygen. Basically, DILITHIUM.KeyGen can be split into 3 phases: the sampling of uniform matrices A, S_1 and S_2; the computation of $T = A \cdot S_1 + S_2$; and the computation of high-order bits of T using the PowerToRound function. Variables S_1 and S_2 are clearly sensitive data because they are part of the secret key what is not the case of variable $T = A \cdot S_1 + S_2$ since it is part of the public key. Consequently, only lines 3 and 4 of Algorithm 1 require masking, i.e. the sampling of S_1 and S_2, usage of these secrets in the computation of T and the secured reconstruction of T. The high-level description of the masked version of DILITHIUM.Keygen is proposed in Figure 3.

The first masked operation is `arith::generate` which provides a secured uniform sampling algorithm within a given bound. The choice of arithmetic masking will ease the following computations: the multiplication of A with masked S_1 can be performed independently on each share of S_1 due to the linearity of the operation with respect to the masking. The second masked operation is `arith::unmask` which securely reconstructs an integer from its shares.

Masking of DILITHIUM.Sign. The most sensitive data used in the signature is Y because it is directly linked with the secret S_2 by the equation $Z = Y + C \cdot S_1$. Since both Z and C are public when a valid signature is produced, the attacker just need to solve a linear system of equations to extract S_2. Variable Z is also critical because in case of a rejection, Z leaks partial information about the secret S_1 as stated in the original security proof of Dilithium. Thus, intermediate Z must be protected.Function H however does not need to be protected. Its inputs ρ, T_1, μ and its output C are public and W_1 is not sensitive (W_1 is reconstructed from public data in the signature verification).

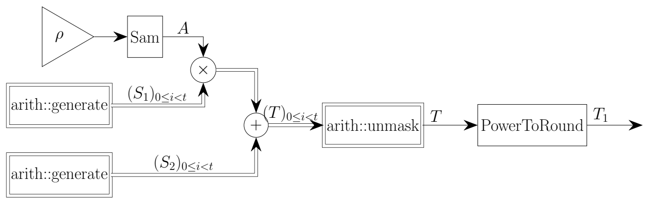


Fig. 3: Masked implementation of DILITHIUM.Keygen. Masked functions are represented with a double lined box.

In Figure 4, we present the masked version of DILITHIUM.Sign. Additional gadgets must be introduced namely:

- **arith::to::bool::lowbits** which securely computes the $\mathrm{LowBits}_{q,2\gamma_2}$ from arithmetic masked shares, and provides the result as boolean masked shares;
- **arith::rejection** and **bool::rejection** which check if the infinity norm of polynomial A is below a constant β for respectively arithmetic and boolean masked shares;
- **arith::makehint** which securely performs the $\mathrm{MakeHint}_{q,2\gamma_2}$ operation on arithmetic masked inputs and returns an unmasked value.

4.3 Description of secured gadgets of Dilithium with prime modulus

In this section, we provide the description of the different masked gadgets for Dilithium with prime modulus. The decomposition and the $\mathrm{MakeHint}_q$ operations are newly introduced gadget while others were introduced in [3].

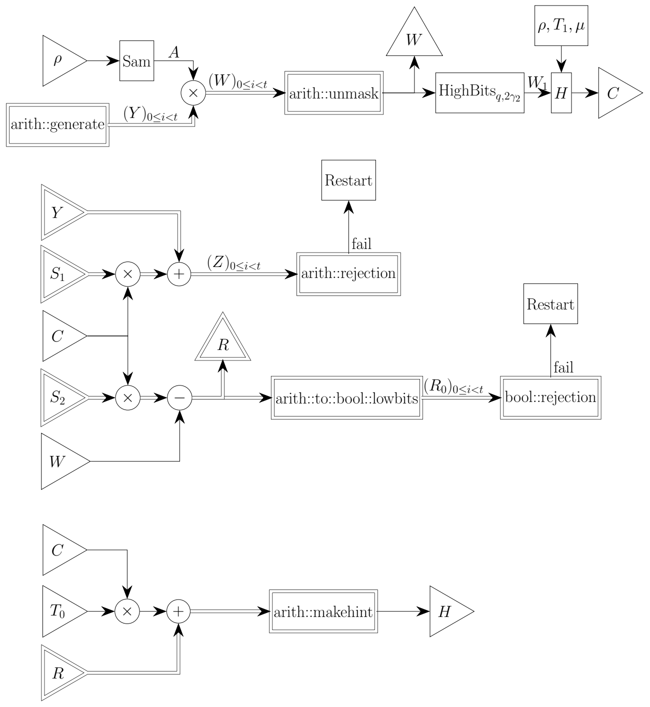


Fig. 4: Masked implementation of DILITHIUM.Sign. Masked functions are represented with a double-lined box.

4.3.1 Description of standard gadgets.

Gadgets are basically split into to categories: linear and non-linear gadgets. Algorithmic definitions of non-linear gadgets can be found in Appendix A.

Linear gadgets can be straightforwardly masked as they are implemented by applying the related instruction separately on each share. Linear gadgets used for the masking of Dilithium are `arith::add` (addition of arithmetic masked shares), `bool::lshift` (left shift of boolean masked shares), `bool::rshift` (right shift of boolean masked shares), `bool::not` (NOT operation on boolean masked shares), `bool::neg` (negation operation on boolean masked shares) and `bool::xor` (XOR operation on boolean masked shares).

Non-linear gadgets are more complex, especially due to the fact that operations between shares are performed implying additional use of randomness (refreshing). Such gadgets are `bool::mask` for the secured masking of a given integer, `arith::to::bool::convert` for the arithmetic to boolean conversion, `bool::add` for the addition on boolean masked shares and `bool::and` for the AND operation on boolean masked shares. These standard gadgets are not a contribution of this paper: for the reader's convenience, a description is given in Appendix A (see Algorithms 9, 8 and 7 respectively).

4.3.2 Description of arith::generate.

The `arith::generate` gadget generates uniformly sampled integers in a given interval. For the non-masked version of Dilithium, this operation is performed in two steps: a first step which uses the XOF function Sam to generate random values; and a second step which checks that the coefficient lies in the target interval and rejects it if not. As stated before, we did not considered the Sam function since the used algorithm may depend on the developers' choice. Since the processing of the generation is analogous to Algorithm 15 of [3] we did not provide full details in the main body, but a description can be found in Appendix A.

4.3.3 Description of arith::rejection and bool::rejection.

The gadget performing the rejection operation on a vector of boolean masked shares called `bool::rejection` is presented in Algorithm 3. For coefficient a, bound β and modulo q, the algorithm checks if $\beta \leq a \leq q - \beta$. The algorithm is constructed by a loop which iterates on all masked coefficients, and evaluates if any coefficient is out of bound by checking both lower and higher bounds.

To do so, the two bound checks are performed by subtracting the given bound to the coefficient and checking the sign bit. It is a similar approach to `arith::generate` at the except that during generation, we only need to check one bound (namely $2 \cdot \beta$) and shift the result by $-\beta$.

The gadget `arith::rejection` is simply implemented as the composition of `arith::to::bool:convert` and `bool::rejection`.

4.3.4 Description of decomposition operations.

Decomposition operations are by far the most complex operations regarding masking. The cornerstone

Algorithm 3 $\texttt{bool::rejection}((\mathbf{a})_{0\le i<t}$,len, $\beta)$

```
1: $(k_0)_{0\le i<t} = \texttt{bool::mask}(-\beta - 1)$
2: $(k_1)_{0\le i<t} = \texttt{bool::mask}(q - \beta - 1)$
3: for $i$ in 0 to $len - 1$
4:     $(b_0)_{0\le i<t} = \texttt{bool::add}((k_0)_{0\le i<t}, (\mathbf{a}[i])_{0\le i<t})$
5:     $(b_0)_{0\le i<t} = \texttt{bool::rshift}((b_0)_{0\le i<t}, 31)$
6:     $(b_1)_{0\le i<t} = \texttt{bool::add}((k_1)_{0\le i<t}, (\mathbf{a}[i])_{0\le i<t})$
7:     $(b_1)_{0\le i<t} = \texttt{bool::rshift}((b_1)_{0\le i<t}, 31)$
8:     $(b_0)_{0\le i<t} = \texttt{bool::xor}((b_0)_{0\le i<t}, (b_1)_{0\le i<t})$
9:     $(r)_{0\le i<t} = \texttt{bool::and}((r)_{0\le i<t}, (b_0)_{0\le i<t})$
10: end for
11: return $\texttt{bool::fullxor}((r)_{0\le i<t})$
```

Algorithm 4 $\texttt{arith:makeint}((r)_{0\le i<t}, (z)_{0\le i<t}, \beta)$. Masked algorithm of $\text{MakeHint}_{q,2\gamma_2}$ with a prime modulus q. w is the word base (usually 32 or 64).

```
1: $(r_1)_{0\le i<t} = \texttt{arith::to::bool::highbits}((r)_{0\le i<t}, \beta)$
2: $(a)_{0\le i<t} = \texttt{arith::addmodq}((r)_{0\le i<t}, (z)_{0\le i<t})$
3: $(a_1)_{0\le i<t} = \texttt{arith::to::bool::highbits}((a)_{0\le i<t}, \beta)$
4: $(t)_{0\le i<t} = \texttt{bool::xor}((r_1)_{0\le i<t}, (a_1)_{0\le i<t})$
6: return $\texttt{bool::fullxor}((t)_{0\le i<t}) \gg (w-1)$
```

is the function $\text{Decompose}_{q,2\gamma_2}$ which takes an integer r as input and returns (r_0, r_1) such that $r = 2r_1\gamma_2 + r_0$. The value r_0 (reps. r_1) is precisely $\text{LowBits}_{q,2\gamma_2}(r)$ (resp. $\text{HighBits}_{q,2\gamma_2}(r)$). Both functions are actually computed using a call to $\text{Decompose}_{q,2\gamma_2}$ then returning the relevant part of r since no relevant optimization can be made when only one of the r_i's is needed.

To illustrate the complexity of this computation, a constant time implementation of $\text{Decompose}_{q,2\gamma_2}$ is provided in Appendix B.1. This algorithm leverages the specific form of both the modulus q and the base used to perform the Euclidean division so that only some shifts and integer additions are used. However, even with these optimizations, $\text{Decompose}_{q,2\gamma_2}$ requires numerous non-linear operations (addition of Boolean shares or Boolean AND). The masked version of $\text{Decompose}_{q,2\gamma_2}$ is provided in Algorithm 13 of Appendix B.1.

4.3.5 Description of arith::makehint.

The computation of $\text{MakeHint}_{q,2\gamma_2}$ strongly relies on decomposition gadgets thus its masking is straightforward as soon as there exists a masked version of $\text{HighBits}_{q,2\gamma_2}$. The masked algorithm for computing $\text{MakeHint}_{q,2\gamma_2}$ is proposed in Algorithm 4.

4.4 Optimization of Dilithium masking for power of two modulus

The main drawback of the prime modulus used in the standard version of Dilithium is the number of non-linear operations required during decomposition operations. As an example, the computation of $\mathrm{LowBits}_{q,2\gamma_2}(W - C \cdot S_2)$ in line 10 of Algorithm 2 requires 12,288 `bool::add` and 4,608 `bool::and` operations.

The choice of a prime modulus q of a specific form is mainly made for efficiency reasons, as it makes number-theoretic transform (NTT)-based polynomial multiplications possible. However, when it comes to the masked scheme, using a power of two modulus q instead speeds up almost all masked gadgets and greatly simplifies the masking of $\mathrm{Decompose}_{q,2\gamma_2}$. Polynomial multiplications then have to be carried out using non-Fourier techniques like Karatsuba, but such techniques turn out to be quite competitive for the parameters of Dilithium.

From a security standpoint, one expects the security level of Dilithium using a power-of-two modulus to be essentially the same as that of the original prime modulus scheme. Indeed, the asymptotic security arguments for the underlying lattice problems Module-LWE and Module-SIS are known to hold for moduli of an arbitrary arithmetic form. This was established by Langlois and Stehlé in their paper on worst-case to average-case reductions for module lattices [15], specifically as Theorem 3.6 for Module-SIS, and Theorem 4.8 (using a modulus switching argument) for Module-LWE. In addition, while in practice parameters are set to match the best concrete lattice attacks on the scheme rather than using security reductions, using a power-of-two modulus does not appear to make any known concrete attack faster compared to the prime modulus case. We also note that power-of-two moduli are commonly used by designers of practice-oriented lattice-based constructions, including the NIST-submitted encryption scheme Saber [7].

Consequently, we propose this power-of-two variant of Dilithium as a relevant alternative insofar as side-channel resistance is a concern.

4.4.1 Simplification of `arith::generate`. The new `arith::generate` is proposed in Algorithm 5. As q is a power of two, and due to the fact that computer units perform two's complement arithmetic, the integer modular reduction after the rejection sampling can be skipped. Moreover, even if the size of the modulus is different from the computer base arithmetic (usually 32-bit of 64-bit), the modular reduction is almost a truncation of high-order bits so we do not need to take into account modular reduction during intermediate computations.

We also found that for the power of two case, it is faster to generate input random integers with arithmetic masked shares (see Section 5). It is not a trivial result because the bound check loop now requires a conversion from arithmetic to boolean masking, and this operation is known to be expensive.

4.4.2 Adaptation of `bool::rejection`. The `bool::rejection` operation is almost unchanged. The only difference is the fact that because the integer modular reduction with a power of two modulus is a truncation of high order

Algorithm 5 `arith::generate`(β). Generates an uniformly sampled integer in the bounds $[-\beta, +\beta]$.

```
1:  mask       $= 1 << (\texttt{NumberOfBits}(\beta) + 1) - 1$
2:  do
3:      for $i$ in 0 to $t - 1$
4:          $(x)_i$       $= \texttt{rand}() \wedge \text{mask}$
5:      end for
6:      $(x)_0$       $= (x)_0 - 2 \cdot \beta - 1$
7:      $(b)_{0 \le i < t} = \texttt{arith::to::bool::convert}((x)_{0 \le i < t})$
8:  while $\texttt{bool::recompose}((b)_{0 \le i < t}) = 0$
9:      $(x)_0$       $= (x)_0 + \beta + 1$
10: return $(x)_{0 \le i < t}$
```

bits, the implementation of the rejection sampling does not require the exact exponent of the modulus q (see line 2 of Algorithm 14 in Appendix C).

4.4.3 Simplification of decomposition operations.

In the Dilithium specification, the decomposition operations are performed in base $2\gamma_2 = \gamma_1 = (q-1)/16$ ($q - 1$ is divisible by 16). Using q a power of two, we have to decompose using a base $2\gamma_2 = 2^b$. Therefore, the decomposition operations become straightforward and are close to a truncation (at the except that the remainder must be zero centered).

Algorithm 6 provides the new constant time implementation of $\text{Decompose}_{q,2\gamma_2}$ with a power of two modulus q (hence a power of two base). As one can see, it is now possible to separate computations of the low order bits and high order bits. This is directly correlated with the fact that q is divisible by 16 (and not $q - 1$) so there is no need to check the border case where $r - r_0 = q - 1$.

An explanation of Algorithm 6 is provided in Appendix B.2. The masked versions of $\text{LowBits}_{q,2\gamma_2}$ (referred as arith::to::bool::lowbits), $\text{HighBits}_{q,2\gamma_2}$ (referred as arith::to::bool::highbits) and $\text{MakeHint}_{q,2\gamma_2}$ (referred as arith::makehint) are presented in Appendix C (Algorithms 15, 16 and 17).

5 Implementation results

In this section, we provide details on the implementation of masking for Dilithium, along with execution times and a side-channel leakage evaluation. The followed approach is similar to the one used for the evaluation of the unprotected implementation in Section 3.

Algorithm 6 Decompose$_{q,2\gamma_2}(r)$.
Parameters: b such that $2^b = 2\gamma_2$ and w the processor word size.

```
1: $m$      $= (1 \ll b) - 1$
2: $d$      $= 1 \ll (b - 1)$
Computation of $r_0$
3: $r_0$    $= r \ll (w - b)$
4: $m_0$    $= \texttt{MaskFromSign}(r_0)$
5: $m_0$    $= m_0 \ll b$
6: $r_0$    $= r_0 \gg (w - b)$
7: $r_0$    $= r_0 \oplus m_0$
Computation of $r_1$
8: $r_1$    $= (r + d) \gg b$
9: return $(r_0, r_1)$
```

5.1 Challenges of the masked implementation

We faced several challenges for the implementation of side channel countermeasures on the ARM Cortex-M3.

The first challenge was the complexity of masking itself. Top level Dilithium gadgets are constructed by calls of common sub-gadgets (which are also possibly large ones). Thus, inlining all procedures were not a relevant approach. Instead, we have evaluated the trade-off between function calls and inlining to reduce memory footprint with a limited impact on performances.

The second challenge was the limitation of the processor micro-architecture. Even with a program following the theoretical t-probing model, the processor micro-architecture itself can possibly leak additional information not covered by the initial model. In the case of the ARM Cortex-M3 micro-architecture, such sensitive components are intermediate registers r_a and r_b which are located between standard registers and arithmetic units (and thus not directly accessible). These registers are not erased between instructions and consequently they leak the transient state of successively manipulated values. Our first implementation in C was actually subject to such leakages and turned out to be unsafe. Thus, we implemented the library in assembly language to control the scheduling of instructions thus overcoming this phenomenon. In addition, since Dilithium gadgets are composed of function calls, we adapted calls to only manipulate addresses of sensitive data instead of the data itself.

A third issue was the complexity of tracking leaky instructions. We first directly evaluated real traces captured with our workbench. However, this approach is time consuming due to trace acquisition and processing. Moreover, the correspondence between timing and assembly instructions is not trivial due to pipelining (it is tractable but takes a lot of time if not automatized). Our final approach was the exploitation of ARM simulators that also evaluate side-channel leakages. We evaluated two of the most recent ones: ELMO [16] and MAPS [6]. Each

Table 1: Execution times of main gadgets for both prime and power of two modulus q on STM32F1 (order-1 masking, computation on 1 coefficient).

	$q = 8380417$	$q = 2^b$	speedup
`arith::to::bool::lowbits`	331 μs / 7,944 cycles	38 μs / 912 cycles	8
`arith::to::bool::highbits`	275 μs / 6,600 cycles	37 μs / 888 cycles	7
`arith::makehint`	560 μs / 13,440 cycles	79 μs / 1,896 cycles	7
`bool::rejection`	66 μs / 1,584 cycles	66 μs / 1,584 cycles	1

Table 2: Execution times of `DILITHIUM.KeyGen` and `DILITHIUM.Sign` on an Intel Core i7-7600U CPU running at 2.80 GHz (10,000 runs).

	Unmasked	Order-1	Order-2	Order-3
`DILITHIUM.KeyGen`	323 μs (*reference*)	1.83 ms (5.66×)	2.52 ms (7.8×)	4.32 ms (13.4×)
`DILITHIUM.Sign`	992 μs (*reference*)	5.64 ms (5.68×)	11.68 ms (11.77×)	28.08 ms (28.3×)

simulator has some idiosyncrasies but for both, the main idea is to simulate the number of bit flips during computations as it is directly correlated to the power consumption. At the time of our experiments, ELMO was only supporting the ARM Cortex-M0 while MAPS was only supporting Cortex-M3. We discuss the relevance of both tools for our particular needs in Appendix D. To take into account the optimization provided by the Cortex-M3, we finally based our simulations on MAPS and brought some modifications to its core to manage some specific instructions.

5.2 Evaluation of execution times

We focused on the most costly masked operations of Dilithium and calculated computation times for both power of two and prime arithmetic. In particular, we have evaluated `arith::to::bool::lowbits`, `arith::to::bool::highbits`, `arith::makehint` and `bool::rejection`. Results are summarized in Table 1.

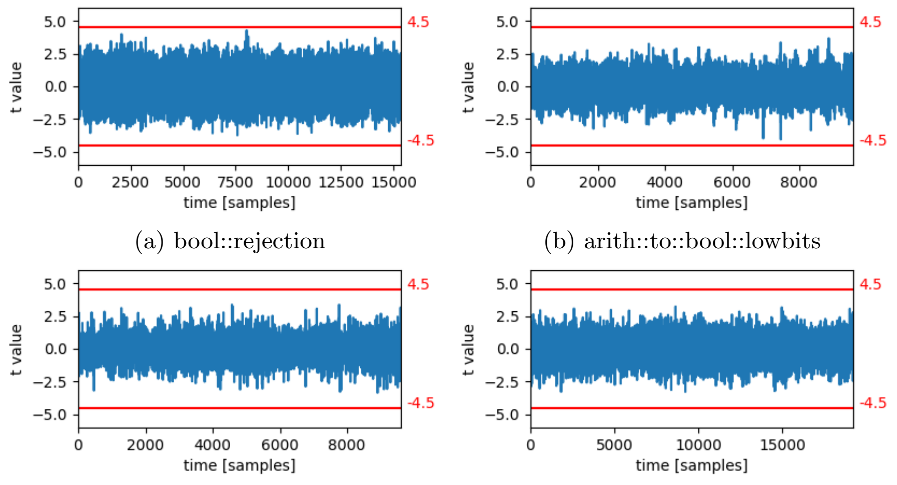
We can observe that the computation times of decomposition operations are greatly improved with power of two modulus, with a speed-up from 7× (for `arith::makehint`) to 8× (for `arith::to::bool::lowbits`). This is due to the fact that only shifts are used for the decomposition when q is a power of two while an Euclidean division is required if q is prime.

We also evaluated the overhead of the masking of Dilithium (power of two implementation) compared to the non-masked version on the full implementation on a general purpose processor. Computation results are summarized in Table 2.

First order masking is 5× slower than unmasked implementation. The complexity of masking is limited due to the possibility of partially masking Dilithium.

5.3 Evaluation of side-channel security

We have evaluated masked gadgets separately due to the limited size on the STM32F1 micro-controller. We focused on the power-of-two modulus version since it corresponds to the main contribution of this paper. To speed up the evaluation phase, we first used MAPS simulator to reduce the majority of leakages. Then, we addressed remaining leakages with our side-channel workbench.



(a) bool::rejection (b) arith::to::bool::lowbits

(c) arith::to::bool::highbits (d) arith::makehint

Fig. 5: Evaluation of the t-test on masked gadgets after 10.000 traces.

In Figure 5, we provide the t-test evaluation of `arith::to::bool::lowbits`, `arith::to::bool::highbits`, `arith::makehint` and `arith::rejection`. We did not detected leakage using 10,000 traces on the first-order protected implementation which is to compare with the high leakages observed using only 500 curves for an unprotected implementation.

References

1. NIST Post-Quantum Cryptography. `http://csrc.nist.gov/groups/ST/post-quantum-crypto/`
2. Albrecht, M.R., Deo, A., Paterson, K.G.: Cold boot attacks on ring and module LWE keys under the NTT. IACR Cryptology ePrint Archive **2018**, 672 (2018)
3. Barthe, G., Belaïd, S., Espitau, T., Fouque, P.A., Grégoire, B., Tibouchi, M.: Masking the GLP lattice-based signature scheme at any order. In: EUROCRYPT (2018)
4. Bindel, N., Buchmann, J., Krämer, J.: Lattice-based signature schemes and their sensitivity to fault attacks. In: FDTC (2016)

5. Chari, S., Jutla, C.S., Rao, J.R., Rohatgi, P.: Towards sound approaches to counteract power-analysis attacks. In: CRYPTO. pp. 398–412 (1999)
6. Corre, Y.L., Großschädl, J., Dinu, D.: Micro-architectural power simulator for leakage assessment of cryptographic software on ARM cortex-m3 processors. In: COSADE. pp. 82–98 (2018)
7. D'Anvers, J., Karmakar, A., Roy, S.S., Vercauteren, F.: Saber: Module-LWR based key exchange, CPA-secure encryption and CCA-secure KEM. In: AFRICACRYPT. pp. 282–305 (2018)
8. Duc, A., Dziembowski, S., Faust, S.: Unifying leakage models: From probing attacks to noisy leakage. In: EUROCRYPT (2014)
9. Ducas, L., Kiltz, E., Lepoint, T., Lyubashevsky, V., Schwabe, P., Seiler, G., Stehlé, D.: Crystals-dilithium: A lattice-based digital signature scheme. IACR Trans. Cryptogr. Hardw. Embed. Syst. **2018**(1), 238–268 (2018)
10. Ducas, L., Kiltz, E., Lepoint, T., Lyubashevsky, V., Seiler, G., Stehlé, D.: CRYSTALS-DILITHIUM, Algorithm Specifications and Supporting Documentation (2017)
11. Espitau, T., Fouque, P.A., Gérard, B., Tibouchi, M.: Loop-abort faults on lattice-based Fiat–Shamir and hash-and-sign signatures. In: SAC (2017)
12. Espitau, T., Fouque, P., Gérard, B., Tibouchi, M.: Side-channel attacks on BLISS lattice-based signatures. In: ACM CCS. pp. 1857–1874 (2017)
13. Groot Bruinderink, L., Hülsing, A., Lange, T., Yarom, Y.: Flush, Gauss, and reload. In: Cryptographic Hardware and Embedded Systems – CHES 2016 (2016)
14. Ishai, Y., Sahai, A., Wagner, D.A.: Private circuits: Securing hardware against probing attacks. In: CRYPTO. pp. 463–481 (2003)
15. Langlois, A., Stehlé, D.: Worst-case to average-case reductions for module lattices. Des. Codes Cryptography **75**(3), 565–599 (2015)
16. McCann, D., Whitnall, C., Oswald, E.: ELMO: emulating leaks for the ARM Cortex-M0 without access to a side channel lab. IACR Cryptology ePrint Archive **2016**, 517 (2016)
17. Pessl, P., Groot Bruinderink, L., Yarom, Y.: To BLISS-B or not to be—attacking strongswan's implementation of post-quantum signatures. In: ACM CCS (2017)
18. Prouff, E., Rivain, M.: Masking against side-channel attacks: A formal security proof. In: EUROCRYPT (2013)

A Standard non-linear gadgets for prime modulus

For the sake of completeness, we provide in this appendix the description of standard gadgets for a prime modulus.

The classical `and` and `add` operations are masked using Boolean shares and are respectively described in Algorithm 7 and Algorithm 8.

Algorithm 7
`bool::and`$((a)_{0\leq i<t}, (b)_{0\leq i<t})$

```
1:  for $i$ in 0 to $t-1$
2:      $(r)_i = (a)_i + (b)_i$
3:  end for
4:  for $i$ in 0 to $t-1$
5:      for $j$ in $i+1$ to $t-1$
6:          $z_{ij} = $ rand()
7:          $z_{ji} = (a)_i \wedge (b)_j$
8:          $z_{ji} = z_{ij} \oplus z_{ji}$
9:          $x = (a)_j \wedge (b)_i$
10:         $z_{ji} = x \oplus z_{ji}$
11:         $(r)_i = (r)_i \oplus z_{ij}$
12:         $(r)_j = (r)_j \oplus z_{ji}$
13:     end for
14: end for
15: return $(r)_{0\leq i<t}$
```

The conversion from arithmetic masking to Boolean masking is shown in Algorithm 9. Is is used in `arith::generate`, and `arith::to::bool::decompose`.

Algorithm 8
`bool::add`$((a)_{0\leq i<t}, (b)_{0\leq i<t})$

```
1:  $(p)_{0\leq i<t}$ = bool::xor($(a)_{0\leq i<t}, (b)_{0\leq i<t}$)
2:  $(g)_{0\leq i<t}$ = bool::and($(a)_{0\leq i<t}, (b)_{0\leq i<t}$)
3:  for $i$ in 1 to $\log(\omega) - 1$
4:      pow         = $1 << (j-1)$
5:      aux         = bool::lshift($(g)_{0\leq i<t}$, pow)
6:      aux         = bool::and($(p)_{0\leq i<t}$, aux)
7:      $(g)_{0\leq i<t}$ = bool::xor($(g)_{0\leq i<t}$, aux)
8:      $\text{aux}_0$        = bool::lshift($(p)_{0\leq i<t}$, pow)
9:      $\text{aux}_0$        = bool::refresh($\text{aux}_0$)
10:     $(p)_{0\leq i<t}$ = bool::and($(p)_{0\leq i<t}, \text{aux}_0$)
11: end for
12: aux         = bool::lshift($(g)_{0\leq i<t}, 1 << (\log(\omega) - 1)$)
13: aux         = bool::and(aux, $(p)_{0\leq i<t}$)
14: $(g)_{0\leq i<t}$ = bool::xor($(g)_{0\leq i<t}$, aux)
15: aux         = bool::lshift($(g)_{0\leq i<t}$, 1)
16: $(r)_{0\leq i<t}$ = bool::xor($(a)_{0\leq i<t}, (b)_{0\leq i<t}$)
17: $(r)_{0\leq i<t}$ = bool::xor($(r)_{0\leq i<t}$, aux)
18: return $(r)_{0\leq i<t}$
```

Algorithm 9 `arith::to::bool::convert`$((a)_{0\leq i<t})$

```
1: $(r)_{0\leq i<t}$ = bool::mask($(a)_0$)
2: for $i$ in 1 to $t-1$
3:         $(x)_{0\leq i<t}$ = bool::mask($(a)_i$)
4:         $(r)_{0\leq i<t}$ = bool::add($(r)_{0\leq i<t}, (x)_{0\leq i<t}$)
5: end for
6: return $(r)_{0\leq i<t}$
```

The `arith::generate` gadget description is proposed in Algorithm 10. Two functions are introduced namely NumberOfBits which is a function to extract the index of the leading positive bit of a given integer, and `bool::maskfromsign` gadget which extracts from a boolean masked integer the sign bit and converts it to a binary all-ones mask (gadget presented in Algorithm 11). We now briefly present the approach of `arith::generate`(β) for a given bound β.

Algorithm 10 `arith::generate`(β). Generates a uniformly sampled integer in the bounds $[-\beta, +\beta]$ in mod q arithmetic. ω is the computer word size (usually 32 bits or 64 bits).

```
1:  $(k)_{0\leq i<t}$ = bool::mask$(-2\beta - 1)$
2:  $(k_0)_{0\leq i<t}$ = bool::mask$(-\beta - 1)$
3:  $(k_1)_{0\leq i<t}$ = bool::mask$(q - 2\beta - 1)$
4:  mask    $= 1 << ($NumberOfBits$(\beta) + 1) - 1$
5:  do
6:      for $i$ in 0 to $t - 1$
7:          $(x)_i$    $=$ rand() $\wedge$ mask
8:      end for
9:      $(b)_{0\leq i<t}$ = bool::add$((x)_{0\leq i<t}, (k)_{0\leq i<t})$
10:     $(b)_{0\leq i<t}$ = bool::rshift$((b)_{0\leq i<t}, \omega - 1)$
11: while bool::recompose$((b)_{0\leq i<t}) = 0$
12: $(b)_{0\leq i<t}$ = bool::add$((x)_{0\leq i<t}, (k_0)_{0\leq i<t})$
13: $(b)_{0\leq i<t}$ = bool::maskfromsign$((b)_{0\leq i<t})$
14: $(b)_{0\leq i<t}$ = bool::and$((b)_{0\leq i<t}, (k_1)_{0\leq i<t})$
15: $(x)_{0\leq i<t}$ = bool::add$((x)_{0\leq i<t}, (b)_{0\leq i<t})$
16: $(r)_{0\leq i<t}$ = bool::to::arith::convert$((x)_{0\leq i<t})$
17: return $(r)_{0\leq i<t}$
```

First, from line 5 to 11, a loop is executed until an integer is sampled in $[-\beta, \beta]$. To do so, we subtract $2 \cdot \beta + 1$ (that is adding k) from a freshly sampled integer in masked form $(x)_{0\leq i<t}$ and check the sign bit. Because we need at some point a shift to check the sign (so a boolean operation), $(x)_{0\leq i<t}$ is generated in boolean masked form.

Second, from line 12 to 15, we determine if $(x)_{0\leq i<t}$ belongs to $[0, \beta]$ or $(\beta, 2 \cdot \beta]$. In the second case, we subtract $2 \cdot \beta$ to shift the result to $[-\beta, 0]$. As operations are in mod q arithmetic, this is equivalent to add $q - 2 \cdot \beta$.

Third, in line 16, a conversion from Boolean to arithmetic masking is performed.

Algorithm 11 `bool::maskfromsign`$((r)_{0\leq i<t})$. Gadget that computes a mask from the sign of a boolean masked shares. ω is the computer word base (usually 32 or 64).

```
1: $(a)_{0\leq i<t} = $ bool::rshift$((r)_{0\leq i<t}, \omega - 1)$
2: $(a)_{0\leq i<t} = $ bool::neg$((a)_{0\leq i<t})$
```

B Details on Decompose$_{q,2\gamma_2}$

B.1 For prime modulus

As outlined in Section 4.3.4, the decomposition function is complex when a prime modulus is used. Algorithm 12 is a constant time algorithm to perform this operation.

Algorithm 12 Decompose(r).

- prime modulus $q = 8380417$
- base $\alpha = 523776 = 2^{19} - 2^9$

```
$r_0 = [r]_\alpha$
 1: $t_0$     $= r \wedge 0x7FFFF$
 2: $t_1$     $= r \gg 19$
 3: $r_0$     $= t_0 + (t_1 \ll 9)$
Ensure that $r_0$ is in range $(-\frac{\alpha}{2}; +\frac{\alpha}{2}]$
 4: $r_0$     $= r_0 - \frac{\alpha}{2} - 1$
 5: $m$       $= \text{MaskFromSign}(r_0)$
 6: $r_0$     $= r_0 + m \wedge \alpha$
 7: $r_0$     $= r_0 - \frac{\alpha}{2} + 1$
Computation of $r_1 = (r - r_0)/\alpha$
 8: $r_1$     $= r - r_0$
 9: $m$       $= (r_1 - 1) \gg 31$
10: $r_1$     $= (r_1 \gg 19) + 1 - m$
Evaluating the specific case $r - r_0 = q - 1$
11: $r_1$     $= r_1 \gg 4$
12: $m$       $= \text{MaskFromSign}(r_1)$
13: $r_1$     $= r_1 \wedge m$
14: $r_0$     $= r_0 - m \wedge 1 + q$
15: return $(r_0, r_1)$
```

When it comes to masking, this complexity bring a non-negligible computational cost. A masked version (taking an arithmetically masked input and providing Boolean masked outputs) is given in Algorithm 13.

Algorithm 13 `arith::to::bool::decompose`$((r)_{0\le i<t})$
Parameters: prime modulus $q = 8380417$ and base $\alpha = 523776$

```
 1: $(m)_{0\le i<t}$= bool::mask($0x7FFFF$)
 2: $(\alpha)_{0\le i<t}$= bool::mask($\alpha$)
 3: $(\alpha_1)_{0\le i<t}$= bool::mask($-(\frac{\alpha}{2}+1)$)
 4: $(\alpha_2)_{0\le i<t}$= bool::mask($-(\frac{\alpha}{2}-1)$)
 5: $(k_0)_{0\le i<t}$= bool::mask($q-1$)
 6: $(k_1)_{0\le i<t}$= bool::mask($1$)
$r_0 = [r]_\alpha$
 7: $(r_p)_{0\le i<t}$= arith::to::bool::convert($(r)_{0\le i<t}$)
 8: $(r_0)_{0\le i<t}$= bool::and($(r_p)_{0\le i<t}, (m)_{0\le i<t}$)
 9: $(m)_{0\le i<t}$= bool::rshift($(r_p)_{0\le i<t}, 19$)
10: $(m)_{0\le i<t}$= bool::lshift($(m)_{0\le i<t}, 9$)
11: $(r_0)_{0\le i<t}$= bool::add($(m)_{0\le i<t}, (r_0)_{0\le i<t}$)
Ensure that $r_0$ is in range $(-\frac{\alpha}{2}; +\frac{\alpha}{2}]$
12: $(r_0)_{0\le i<t}$= bool::add($(r_0)_{0\le i<t}, (\alpha_1)_{0\le i<t}$)
13: $(m)_{0\le i<t}$= bool::rshift($(r_0)_{0\le i<t}, 31$)
14: $(m)_{0\le i<t}$= bool::neg($(m)_{0\le i<t}$)
15: $(m)_{0\le i<t}$= bool::and($(m)_{0\le i<t}, (\alpha)_{0\le i<t}$)
16: $(r_0)_{0\le i<t}$= bool::add($(r_0)_{0\le i<t}, (m)_{0\le i<t}$)
17: $(r_0)_{0\le i<t}$= bool::add($(r_0)_{0\le i<t}, (\alpha_2)_{0\le i<t}$)
Computation of $r_1 = (r - r_0)/\alpha$
18: $(r_1)_{0\le i<t}$= bool::not($(r_0)_{0\le i<t}$)
19: $(r_1)_{0\le i<t}$= bool::add($(r_1)_{0\le i<t}, (r_p)_{0\le i<t}$)
20: $(u)_{0\le i<t}$= bool::rshift($(r_1)_{0\le i<t}, 31$)
21: $(u)_{0\le i<t}$= bool::neg($(u)_{0\le i<t}$)
22: $(r_1)_{0\le i<t}$= bool::add($(r_1)_{0\le i<t}, (k_1)_{0\le i<t}$)
23: $(r_1)_{0\le i<t}$= bool::rshift($(r_1)_{0\le i<t}, 19$)
24: $(u)_{0\le i<t}$= bool::not($(u)_{0\le i<t}$)
25: $(u)_{0\le i<t}$= bool::lshift($(u)_{0\le i<t}, 31$)
26: $(u)_{0\le i<t}$= bool::rshift($(u)_{0\le i<t}, 31$)
27: $(r_1)_{0\le i<t}$= bool::add($(r_1)_{0\le i<t}, (u)_{0\le i<t}$)
Evaluating the specific case $r - r_0 = q - 1$
28: $(m)_{0\le i<t}$= bool::lshift($(r_1)_{0\le i<t}, 32-4-1$)
29: $(m)_{0\le i<t}$= bool::rshift($(m)_{0\le i<t}, 31$)
30: $(m)_{0\le i<t}$= bool::neg($(m)_{0\le i<t}$)
31: $(m)_{0\le i<t}$= bool::not($(m)_{0\le i<t}$)
32: $(r_1)_{0\le i<t}$= bool::and($(r_1)_{0\le i<t}, (m)_{0\le i<t}$)
33: $(m)_{0\le i<t}$= bool::lshift($(m)_{0\le i<t}, 31$)
34: $(m)_{0\le i<t}$= bool::rshift($(m)_{0\le i<t}, 31$)
35: $(r_0)_{0\le i<t}$= bool::add($(r_0)_{0\le i<t}, (m)_{0\le i<t}$)
36: $(r_0)_{0\le i<t}$= bool::add($(r_0)_{0\le i<t}, (k_0)_{0\le i<t}$)

37: return $((r_0)_{0\le i<t}, (r_1)_{0\le i<t})$
```

B.2 For power of two modulus

In Section 4.4, the masked version of the decomposition function is outlined (Algorithm 6). We propose here a more detailed explanation to convince the reader that this algorithm actually computes the decomposition.

First, a binary truncation is performed on r. Figure 6 provides the binary representation of numbers for this operation. At this point, the truncation produces r'_0 and r'_1 such as $r = r'_0 + \alpha r'_1$ and $r'_0 \in [0, \alpha)$.

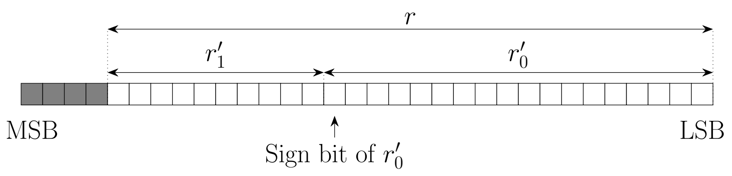


Fig. 6: Binary representation of numbers r, r'_0 and r'_1 such as $r = r'_0 + r'_1 \cdot \alpha$, with α a power of two and $r'_0 \in [0, \alpha)$

To shift r'_0 to the interval $[-\alpha, \alpha)$, we only need to expand the sign bit of r'_0 as shown in Figure 6. To do so, we first shift the sign bit to the most significant bit, negate the result to create a mask then set to zero the $\log_2 \alpha - 1$ bits with two successive shifts (line 3 to 6 of Algorithm 6).

If r'_0 is negative, then we must add 1 to r'_1. As the sign bit of r'_0 can be directly determined from r, we do not need to compute r'_0 to compute r_1. To do so, we perform the addition between r and a number d constructed such as all bits are set to 0 except at the sign bit of r'_0. If the sign bit of r'_0 is 0, then $r + d$ does not produce carry propagation on the high order bits of r and so does not modify r'_1. If the sign bit of r'_0 is 1, then there is a carry propagation which corresponds to adding 1 to r'_1. So in both cases, we have computed the wright r_1. Finally, a last shift is performed in order to keep only high order bits of $r + d$.

C Gadgets for Dilithium with power of two modulus

In this appendix we provide algorithmic descriptions of the gadgets linked to the decomposition process when the modulus is a power of two.

Algorithm 14 `bool::rejection`$((\mathbf{a})_{0\leq i<t}, len, \beta)$

```
1: $(k_0)_{0\leq i<t} =$ bool::mask$(-\beta - 1)$
2: $(k_1)_{0\leq i<t} =$ bool::mask$(\beta - 1)$
3: for $i$ in 0 to $len - 1$
4:         $(b_0)_{0\leq i<t} =$ bool::add$((k_0)_{0\leq i<t}, (\mathbf{a}[i])_{0\leq i<t})$
5:         $(b_0)_{0\leq i<t} =$ bool::rshift$((b_0)_{0\leq i<t}, 31)$
6:         $(b_1)_{0\leq i<t} =$ bool::add$((k_1)_{0\leq i<t}, (\mathbf{a}[i])_{0\leq i<t})$
7:         $(b_1)_{0\leq i<t} =$ bool::rshift$((b_1)_{0\leq i<t}, 31)$
8:         $(b_0)_{0\leq i<t} =$ bool::xor$((b_0)_{0\leq i<t}, (b_1)_{0\leq i<t})$
9:         $(r)_{0\leq i<t} =$ bool::and$((r)_{0\leq i<t}, (b_0)_{0\leq i<t})$
10: end for
11: return bool::fullxor$((r)_{0\leq i<t})$
```

Algorithm 15 `arith::to::bool::lowbits`$((r)_{0\leq i<t}, \beta)$.
Parameters: b such that $2^b = q$ and w the processor word size.

```
1: $(r_0)_{0\leq i<t} =$ arith::to::bool::convert$((r)_{0\leq i<t})$
2: $(r_0)_{0\leq i<t} =$ bool::lshift$((r_0)_{0\leq i<t}, \omega - \log_2 \beta)$
3: $(b)_{0\leq i<t} =$ bool::maskfromsign$((r_0)_{0\leq i<t})$
4: $(b)_{0\leq i<t} =$ bool::lshift$((b)_{0\leq i<t}, \log_2 \beta)$
5: $(r_0)_{0\leq i<t} =$ bool::rshift$((r_0)_{0\leq i<t}, \omega - \log_2 \beta)$
6: $(r_0)_{0\leq i<t} =$ bool::xor$((r_0)_{0\leq i<t}, (b)_{0\leq i<t})$
7: return $(r_0)_{0\leq i<t}$
```

Algorithm 16 `arith::to::bool::highbits`$((r)_{0\leq i<t}, \beta)$.
Parameters: b such that $2^b = q$ and w the processor word size.

```
1: mask $= \beta - 1$
2: $(d)_{0\leq i<t}$ = arith::mask((mask >> 1) + 1)
3: $(r_1)_{0\leq i<t}$ = arith::add($(r)_{0\leq i<t}$, $(d)_{0\leq i<t}$)
4: $(r_1)_{0\leq i<t}$ = arith::rshift($(r_1)_{0\leq i<t}$, $\log_2 \beta$)
5: return $(r_1)_{0\leq i<t}$
```

Algorithm 17 `arith::makeint`$((r)_{0\leq i<t}, (z)_{0\leq i<t}, \beta)$.
Parameters: b such that $2^b = q$ and w the processor word size.

```
1: $(r_1)_{0\leq i<t}$ = arith::to::bool::highbits($(r)_{0\leq i<t}$, $\beta$)
2: $(a)_{0\leq i<t}$ = arith::add($(r)_{0\leq i<t}$, $(z)_{0\leq i<t}$)
3: $(a_1)_{0\leq i<t}$ = arith::to::bool::highbits($(a)_{0\leq i<t}$, $\beta$)
4: $(t)_{0\leq i<t}$ = bool::xor($(r_1)_{0\leq i<t}$, $(a_1)_{0\leq i<t}$)
5: $(t)_{0\leq i<t}$ = bool::lshift($(r_1)_{0\leq i<t}$, $(r_1)_{0\leq i<t}$, $w - 1$)
6: $c$ = bool::fullxor($(t)_{0\leq i<t}$)
7: return $c$
```

D Choice of an ARM leakage simulator

In the case of ELMO simulator, the power estimation is based on a template made from real traces of the Cortex-M0. The power consumption is estimated by evaluating bit flips, Hamming weights and Hamming distances of operands between the previous, the current and the subsequent operation.

On the other hand, authors of MAPS had access to the RTL of the Cortex-M3 leading to a different simulation strategy. The simulator thus takes into account some hidden features as the state of pipeline registers (in particular, registers r_a and r_b). Then, when the state of a register changes, the number of bit flips is computed to estimate the power consumption, and the result is pushed to the output. Consequently, the estimator is not cycle accurate as registers are evaluated one after the other. MAPS only provides leakage of core registers, thus leakage from peripherals or the ALU are not considered.

We faced several limitations when evaluating the masking of Dilithium. First, ELMO, which has been designed for Cortex-M0, does not target the same micro-controller than our workbench, i.e. the STM32F1 micro-controller (which is a Cortex-M3). The main difference between Cortex-M0 and Cortex-M3 is the fact that Cortex-M0 has very limited access to Thumb 2 instructions (32 bit instructions), leading in the majority of cases to two different implementations. Second, both ELMO and MAPS were initially developed for symmetric cryptography, which essentially requires less resources than Dilithium (both in term of memory, time and instruction set). For ELMO, which basically stores information about each instruction in RAM, we run out of memory during the execution of larger gadgets like arith::to::bool::lowbits or bool::rejection. This issue can be addressed by predicting data stored in RAM into a file when needed. For MAPS, we had to modify the core library to extend the instruction set implemented, as the simulator faced several unknown instructions (i.e. not yet implemented). In particular, the branch with link instruction, which allow to branch the program to a sub-routine, were not implemented but was critical in Dilithium masking as gadgets cannot be entirely inlined due to their complexity. To take into account the optimization provided by the Cortex-M3, we finally based our simulations on MAPS.