Masking Dilithium: Efficient Implementation and Side-Channel Evaluation

Vincent Migliore[1], Benoit 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 postquantum 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 postquantum 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, in particular, with a careful 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

Postquantum 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 postquantum secure primitives gets more pressing. In particular, there is a growing push to make postquantum cryptography, which was of somewhat theoretical interest for some time, ready for real-world deployment. At the forefront of that push is NIST's postquantum standardization process [1], which aims at selecting postquantum 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 [4,10], cold boot attacks [2], cache timing attacks [12,15] and more standard power/electromagnetic analysis [11], 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. [8], a NIST candidate that can be seen as a descendent 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 microcontroller (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;
4. we implement these masked schemes on the same ARM Cortex-M3 microcontroller, and evaluate both their efficiency and side-channel resistance, with satisfactory results on both counts.

The paper is organized as follows. Section **??** recalls the key generation and the signing algorithms of Dilithium. Section 4 evaluates the side-channel leakage of sensitive operations on our STM32F1 target microcontroller. Section 5 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 6 provides implementation results, both in terms of performance and of side-channel resistance. Finally, Section 7 presents our conclusions.

2 Notation

Let $\mathbb{Z}_q[X] = (\mathbb{Z}/q\mathbb{Z})[X]$ be the set of polynomials with integer coefficients modulo q. We define $R_q = \mathbb{Z}_q[X]/(X^n + 1)$ the ring of polynomials with integer coefficients modulo q, reduced by the cyclotomic polynomial $X^n + 1$. For any $\alpha \in \mathbb{N}$, $\alpha < q$, R_α refers to the set of polynomials with coefficients in $[-\frac{\alpha}{2}; +\frac{\alpha}{2}]$. In the following, all polynomial operations are considered performed in R_q.

A polynomial is represented with an uppercase and its coefficients with a lowercase. For polynomial A, a_i represents its i^{th} coefficient. For coefficient a_i of polynomial A, $a_{i,(k\leftarrow j)}$ corresponds to the binary string extraction of a_i between bits j and k. This notation is extended to polynomial A where $A_{(k\leftarrow j)}$ is the sub-polynomial where the binary string extraction is applied to each coefficient.

For a set R of polynomials and a polynomial A, $A \leftarrow U_R$ represents an uniformly sampled polynomial in R and $A \leftarrow B_R$ a uniformly sampled polynomial in R with binary coefficients.

A modular reduction by an integer q is noted $[\cdot]_q$, and $[\cdot]_q^{\pm}$ a modular reduction where the (unique) corresponding result lands in $(-\frac{q}{2}; +\frac{q}{2}]$. For polynomial A and integer a_i, we denote by $||a_i||_\infty = |[a_i]_q^{\pm}|$ and $||A||_\infty = \max_i ||a_i||_\infty$.

For integer a, $\lfloor a \rfloor$, $\lceil a \rceil$ and $\lfloor a \rceil$ operators are respectively the floor, ceil and nearest rounding operations. This notation is extended to polynomials by applying the operation on each coefficient. For vectors A and B, $\langle A, B \rangle$ represents $\sum A[i]B[i]$.

We also note $(a)_{0\leq i<t}$ (resp. $(A)_{0\leq i<t}$) the masked form of integer a (resp. polynomial A) at order t. The i^{th} share of $(a)_{0\leq i<t}$ is noted $(a)_i$. This notation is extended to polynomials, where $(A)_i$ corresponds to the polynomial constructed by extracting the i^{th} share of each coefficient of A.

3 The Dilithium Signature scheme

LWE, R-LWE, Module-LWE. We briefly recap fundamental information about the security foundation of Dilithium, i.e. the LWE problem. Introduced in [17] by Regev, LWE problem consists in recovering a secret s from m samples of the form $(a^{(n)}, \langle a^{(n)}, s^{(n)} \rangle + e \mod q)$, where $a^{(n)}$ is uniformly generated in $\mathbb{Z}_q^n$, $e \leftarrow \chi$ and $s \leftarrow \phi$ with ϕ distribution over $\mathbb{Z}_q^n$ and χ distribution over $\mathbb{Z}_q$. Usually, χ is set to a discrete Gaussian distribution of a given standard deviation σ, and ϕ a very narrow discrete Gaussian distribution that can be reduced to uniform sampling in $\{-1, 0, 1\}$. LWE is known to be as hard as standard (worst-case) problems on euclidean lattices, but come up with a major drawback since arithmetic computations are matrix-matrix and matrix-vector of integer.

A practical variant of LWE is Ring-LWE. The key idea is to sort samples $a^{(n)}$ in such a way that matrix computations can be seen as a polynomial operations of degree n. Samples can be represented as a degree-n polynomial A, providing a more compact implementation. This is the usual instantiation of various LWE-based schemes.

Dilithium is not directly instantiated with R-LWE, but with Module-LWE. With Module, samples are represented as a matrix of small polynomials A instead of a unique polynomial A. This approach addresses a slight limitation of R-LWE: the size of polynomials increases with security. For Module, only the number of rows (noted k) and columns (noted ℓ) impacts security, not the size of polynomials that can be set the same for all instantiations (256 coefficients in Dilithium).

Dilithium signature scheme. Dilithium is a signature scheme based on the Fiat-Shamir with Abort framework and is implemented with Module. Core functions are composed of *KeyGen* for the key generation, *Sign* to produce a signature of a message, and *Verify* to verify the signature.

One of the main breakthrough of Dilithium (aside from the Module approach) is the key compression mechanism to reduce public key size. The compression is performed at two different levels. First, Module sample A is producted 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 A. Second, for the second part of the public key (which is a vector of polynomials), a per-coefficients truncation is performed, associated with a correcting code mechanism to guess truncated bits.

For a formal description of the different truncation procedures used in Dilithium, i.e. Decompose_q / HighBits_q / LowBits_q / Power2Round, reader can refer to the original Dilithium paper [9].

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

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: $T = A \cdot S_1 + S_2 \in \times R_q^{k\times 1}$
5: $T_1 = \mathrm{Power2Round}(T, d) \in \times R_q^{k\times 1}$
6: $P_{key} = (\rho, T_1)$
7: $S_{key} = (\rho', S_1, S_2, T)$
8: return $(P_{key}, S_{key})$
```

In this paper, we mainly focus on the key generation and the signature (which will respectively be called DILITHIUM.KeyGen and DILITHIUM.Sign) since *Verify* does not manipulate sensitive data and does not require proper masking.

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

The randomness generation is computed using an extendable output function (XOF) called Sam (lines 2 to 3), which takes as input a random seed, and outputs an extendable pseudo-random string. The Sam function is used to compute the matrix A (part of the public key) and (S_1,S_2) (part of the secret key). Coefficients of S_1 and S_2 are small coefficients, while coefficients of A are full size.

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. The algorithm is constructed by a rejection sampling loop where a fresh signature is generated until it satisfies some security properties (lines 11 to 13). First of all, a uniformly sampled matrix Y in $R_{\gamma_1 - 1}$ 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 by executing 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 a message.

To ensure that the signature does not leak information about the key, lines 11 to 13 execute 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 required before producing a good set of parameters. Eventually, the MakeHint$_q$ procedure line 14 will produce some hints to help guessing the shrunk bits of the public key.

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:   $W_1 = \mathrm{HighBits}_q(W, 2\gamma_2) \in R_q^{k\times 1}$
 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:   $R_0 = \mathrm{LowBits}_q(W - CS_2, 2\gamma_2)$
11:   if $||Z||_\infty \geq \gamma_1 - \beta$ goto 4
12:   if $||R_0||_\infty \geq \gamma_2 - \beta$ goto 4
13:   if $||CT_0||_\infty \geq \gamma_2$ goto 4
14: $H = \mathrm{MakeHint}_q(-CT_0, W - CS_2 + CT_0, 2\gamma_2)$
15: $\sigma = (Z, H, C)$
16: return $\sigma$
```

4 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. As we aim at putting forward weaknesses of an unprotected implementation, we performed Welch's T-test to localize potential leakages and single-bit DPA on secret variables to confirm that the observed peaks actually correspond to exploitable leakages. We focused our efforts on three critical functions: $\mathrm{LowBits}_q$, $\mathrm{HighBits}_q$ and rejection operations.

Operation choice motivation. The rejection is one of the most critical operations as it is used both for secret data generation and for rejection sampling in the signature loop. For example, with a successful attack on the rejection operation, an attacker can possibly extract partial information on S_1, S_2 during the key generation, Y for the signature or even on a rejected Z (which leaks information about S_1 as explained in the original Dilithium paper). Regarding decomposition operations, we evaluated the $\mathrm{LowBits}_q(W - C \cdot S_2, 2\gamma_2)$ in line 10 of Algorithm 2, and $\mathrm{HighBits}_q$ which is part of the computation of $\mathrm{MakeHint}_q(-CT_0, W - CS_2 + CT_0, 2\gamma_2)$.

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. The situation is similar for the random oracle H as its actual implementation from the NIST submission relies on

SHAKE256 but it is not mandatory. Studying the resistance of these primitives is indeed of great importance before deploying a solution but is somehow out of the scope of this paper.

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. The data was generated on the desktop computer then sent to the DUT which processed it using the targeted operation. A trigger helped the synchronization of the oscilloscope and the DUT through one of the GPIO pins of the discovery board. The EM traces were collected and packed with the corresponding data into an H5 file. Finally, 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 choosing if the next input was the fixed value or a random one. The single-bit DPA has been performed on each bit of the sensitive data in the input of the target operations.

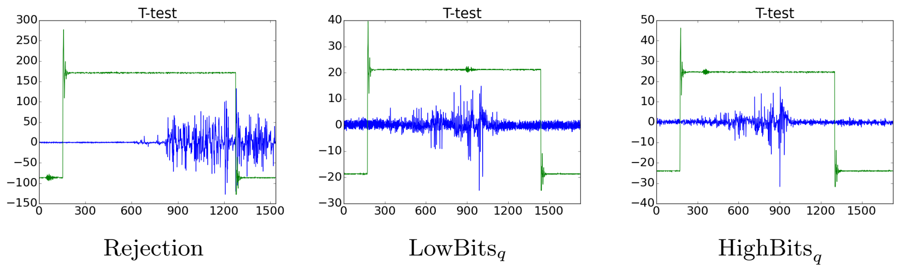


Fig. 1: T-Test evaluation for targeted operations (using 500 traces).

Evaluation results. 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.

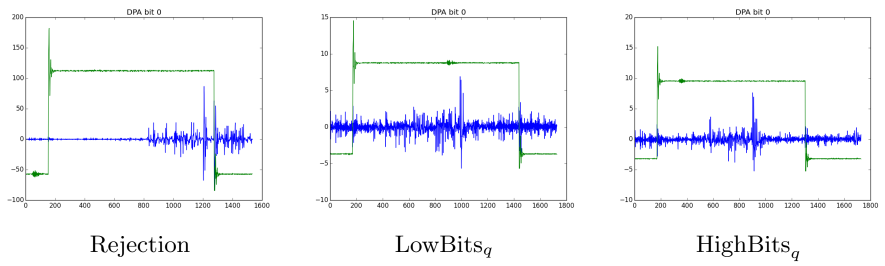


Fig. 2: Single-bit DPA curves on bit 0 of sensitive data (using 500 traces).

5 Masking Dilithium

Results of Section 4 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.

5.1 Leakage model

The first introduced side-channel security model was the noisy leakage model in which the attacker obtains sensitive information mixed with noise [5,16]. The main limitation of this approach is the deep knowledge of the noise it requires which is strongly device-dependent.

A more practical approach is the probing model [13]. 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 [7], a reduction have been obtained proving that security in the t-probing model implies security in the noisy leakage one.

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. With $t+1$ shares, variables are considered masked at order t. The threshold probing model introduces the notion of *t-probing secure gadget*.

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

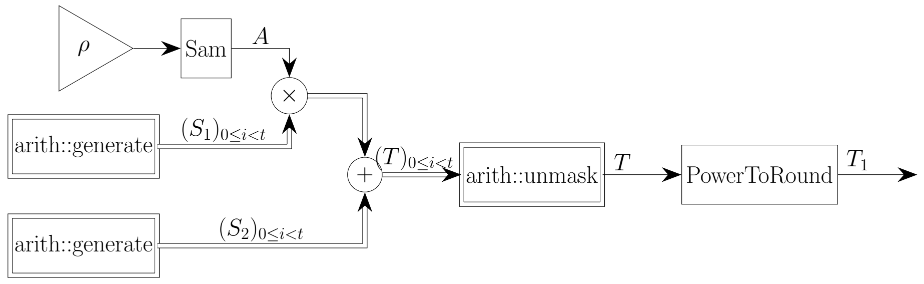


Fig. 3: Masked implementation of DILITHIUM.Keygen. Masked functions are represented with a double ligned box.

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

5.2 Presentation of the masked key generation and signature

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 requires 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. Masked operations are represented with double lined edges.

The first masked operation is arith::generate which provides a secured uniform sampling algorithm within a given bound. The prefix arith:: means that the algorithm manipulates arithmetic masked shares (both input and output variables). This choice 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. Thus no additional randomness is required. This is why the multiplication and the sum are not double-edged in Figure 3.

The second masked operation is arith::unmask which securely reconstructs an integer from its shares.

Masking of DILITHIUM.Sign. 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 LowBits_q from arithmetic masked shares, and provides the result on boolean masked shares;

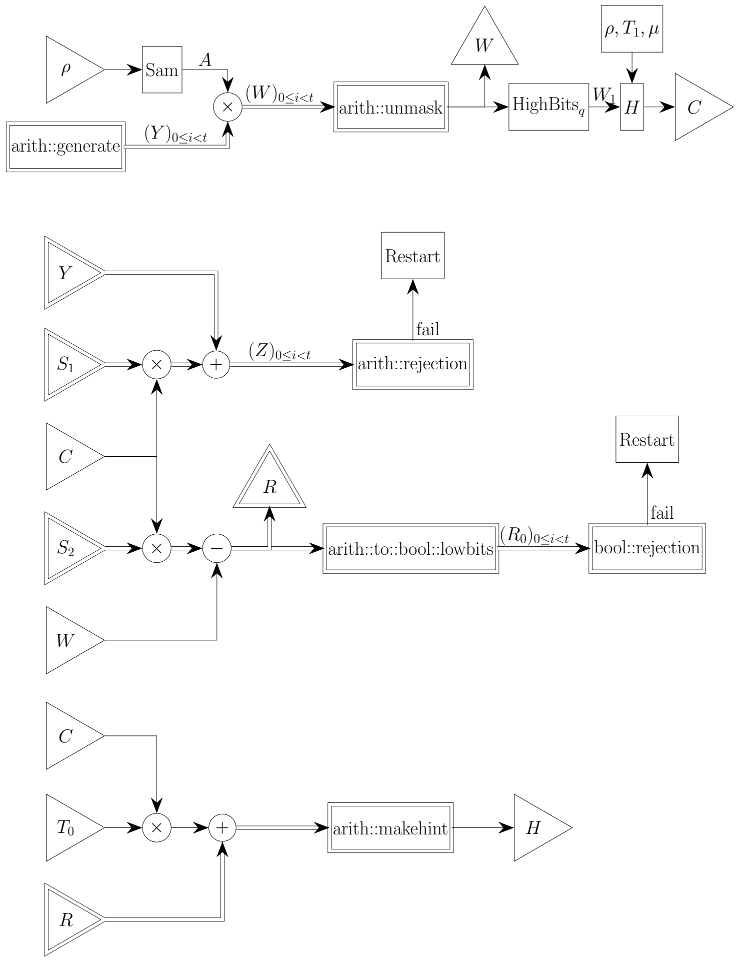


Fig. 4: Masked implementation of DILITHIUM.Sign. Masked functions are represented with a double ligned box.

- **arith::rejection** and **bool::rejection** which are evaluated if the infinity norm of polynomial A is below a constant β for respectively arithmetic masked shares and boolean masked shares;
- **arith::makehint** which securely computes the MakeHint$_q$ operation with a public output.

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).

5.3 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 Decompose_q and MakeHint_q operations are newly introduced gadget while others where introduced in [3].

5.3.1 Description of standard gadgets. Gadgets are basically split into to categories: linear and non-linear gadgets.

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 (Algorithm 6) for the arithmetic to boolean conversion, bool::add (Algorithm 7) for the addition on boolean masked shares and bool::and (Algorithm 8) for the AND operation on boolean masked shares.

5.3.2 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 is in the wright bound and reject if it is not.

The masking of Sam is hard and not particularly efficient. The internal permutation requires 12 loops of 50 (non-linear) AND operations. As a consequence, we didn't mask the Sam function, but instead we picked random number from a random generator. This choice does not impact one of the most interesting feature of Dilithium, i.e. key compression, because public information (public key and the signature) are still compressed. The main drawback is the fact that with a XOF function, Dilithium scheme can be turned to a full deterministic version. In particular, in the Dilithium specification paper proposed for the NIST competition in [8], they reused ρ, T_1, the message μ and W_1 at different levels of the signature scheme to generate fresh seeds.

The arith::generate gadget description is proposed in Algorithm 11, and introduces two functions: NumberOfBits 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 mask (gadget presented in Algorithm 5). We now briefly present the approach of arith::generate(β) for a given bound β.

First, from line 5 to 9, a loop is executed until an integer is sampled in $[-\beta, \beta]$. To do so, we subtract $2 \cdot \beta + 1$ 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 10 to 13, 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 14, a conversion from Boolean to arithmetic masking is performed. As Boolean to arithmetic masking conversion is one of the most costly operation in the algorithm, another approach would have been to generate arithmetic masked integer in the rejection loop. However, the fact that we need at least two bound checks (at least one during the rejection loop, and another one after the loop), we would need at least two arithmetic to Boolean conversions, and so we would be less efficient than the first approach. In the following, we will present a modified version of Dilithium where this last approach is of interest.

5.3.3 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 10. 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. More formally, for coefficient a, bound β and q the Dilithium modulus, the algorithm checks if $\beta \leq a \leq q - \beta$.

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 $-\beta$.

As the rejection of a coefficient is objectively rare (it only occurs 4 times on average)we also masked the intermediate result of the bound check ($(r)_{0 \leq i < t}$ in Algorithm 10). If we did not masked the intermediate result of the bound check, we would expect a very slightly improvement on performances, as we can avoid the bool::and operation in line 9, but its computation time is almost negligible compared to bool::add computation time. The only non-negligible possible speed-up would be the early abort, as breaking the loop before the end saves computation time on the unnecessary check of remaining coefficients.

5.3.4 Description of decomposition operations. Decomposition operations are by far the most complex operations regarding masking in our masked implementation of Dilithium. To illustrate this complexity, Algorithm 4 in appendix provides a constant time implementation of Decompose$_q$. This algorithm leverages the specific form of both the modulus q and the base to perform the Euclidean division with only some shifts and integer additions. However, even with these intensive optimizations, the Decompose_q requires numerous non-linear operations such as the addition of Boolean shares and Boolean AND. Moreover, it is not simple to exploit the fact that only lower or higher order bits are required as there is many dependencies between them. Consequently, it is almost needed to compute Decompose_q and only keep the required bits.

The masked version of Decompose_q is provided in Algorithm 12.

5.3.5 Description of arith::makehint.

As long as masking gadgets of decomposition operations are designed, the masking of MakeHint_q is straightforward as it only requires to run the HighBits_q operations two times and XOR the result. The masked algorithm of MakeHint_q is proposed in Algorithm 9.

5.4 Optimization of Dilithium masking for power of two modulus

The main drawback of prime modulus is the number of non-linear operations required during decomposition operations. As en example, the computation of $\text{LowBits}_q(W - C \cdot S_2, \gamma_2)$ in line 10 of Algorithm 2, 12.288 bool::add and 4.608 bool::and or required. Since polynomial multiplications are usually performed with NTT, the choice of a prime modulus q is essential to maximize NTT efficiency. However, the original Learning With Error problem does not impose a particular form on q.

Consequently, we propose to reduce the complexity of the Dilithium masking by taking the modulus q as a power of two. This consideration almost speed-up all masked gadgets and greatly simplify the masking of Decompose_q. Regarding security, and considering a modulus switching argument, one expects the security level of this power-of-two modulus variant to be essentially the same as that of the original prime modulus scheme.

5.4.1 Simplification of arith::generate.

The new arith::generate is proposed in Algorithm 13. 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 than the base of the computer arithmetic (usually 32 bits of 64 bits), 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 6. 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 costly.

5.4.2 Simplification of arith::rejection.

The arith::rejection operation (presented in Algorithm 16) is almost unchanged but it was not particularly costly even with prime modulus. The only difference is the fact that because the integer modular reduction with a power of two modulus is a truncation of high order bits, the implementation of the rejection sampling does not require the exact exponent of the modulus q.

5.4.3 Simplification of decomposition operations. In the Dilithium specification, the decomposition operations are performed in base $\gamma_1 = (q-1)/16$ ($q-1$ is divisible by 16) and $\gamma_2 = \gamma_1/2$. Using $q = 2^b$, we have to decompose to power of two bases $\gamma_1 = 2^{b-4}$ and $\gamma_2 = 2^{b-5}$. Therefore, the decomposition operations become straightforward and are close to a truncation (at the except that the remainder must be zero centered).

Algorithm 3 Decompose(r).

- power of two modulus q
- power of two base 2^b
- w computer arithmetic base (usually 32 or 64).

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

Algorithm 3 provides the new constant time implementation of Decompose_q with a power of two modulus q. 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 $r - r_0 = q - 1$.

We now briefly present how efficiently implement the decomposition operation of $(r_0, r_1) = \text{Decompose}(r)$ for some integer r, with α the base (which is power of two), r_0 the low order bits of r and r_1 its high order bits.

First, a binary truncation is performed on r. Figure 5 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 $r'_0 \in [0, \alpha)$.

To shift r'_0 to the interval $[-\alpha, \alpha)$, we only need to expand the sign bit of r'_0 as shown in Figure 5. To do so, we first shift the sign bit to the most significant

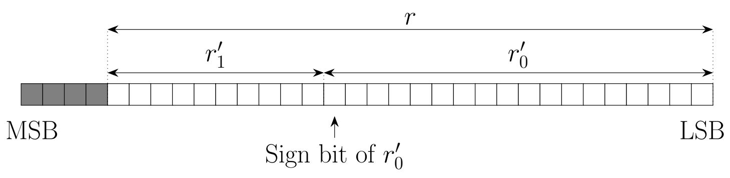


Fig. 5: Binary representation of numbers r, r'_0 and r'_1 such as $r = r'_0 + r'_1 \cdot \alpha$, with α a power of two and $r'_0 \in [0, \alpha)$

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 3).

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 $r + d$.

In Appendix D, Algorithm 14 and Algorithm 15 provide respectively the masked version of LowBits_q (referred as arith::to::bool::lowbits) and HighBits_q (referred as arith::to::bool::highbits).

6 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.

6.1 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 tradeoff between function calls and inlining to reduce memory footprint with a limited impact on performances. As an illustration we present the case of the gadget bool.add. It is a building block of larger gadgets and thus will be called several times. Moreover, the function call execution time is negligible with respect to the execution of the gadget itself. Consequently, bool.add was a good candidate for an encapsulation as a function.

The second challenge was the limitation of the processor architecture. Even with a program following the theoretical t-probing model, the processor architecture itself can possibly leak additionnal 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 to take this phenomenon into account. 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 (carefully managing reads and writes to memory).

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 pipeline (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 [14] and MAPS [6].

6.2 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 evaluated.

We faced several limitations when evaluating the masking of Dilithium. First, ELMO, which has been designed for cortex-M0, does not target the same microcontroller than our workbench, i.e. the STM32F1 microcontroller (which is a cortex-M3). Second, both ELMO and MAPS were initially developped for symmetric cryptography, which essentially requires less resources than Dilithium (both in term of memory, time and instruction set). 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.

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).

	No-masking	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×)

6.3 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. Computation 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 directly correlated with the fact that for power of two implementation, we basically only need shifts for the decomposition, while we need an Euclidean division for prime arithmetic.

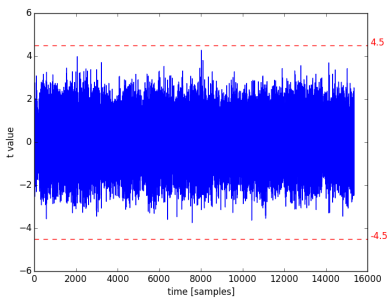
These computation times must be multiplied by the number of coefficients, typically $n \cdot l$ (1024) or $n \cdot k$ (1280), to evaluate the real impact on performances. Considering that a rough estimate of the number of cycles to execute a polynomial multiplication of degree 256 is about 60,000 cycles in an ARM microcontroller, the overhead of prime arithmetic is about 120 polynomial multiplications, so cleary dominate the computation time of Dilithium operations. Consequently, the choice of power of two arithmetic greatly improve the complexity of masking for Dilithium.

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.

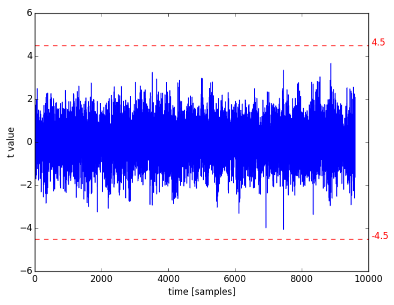
6.4 Evaluation of side-channel security

We have evaluated masked gadgets separately due to the limited size on the STM32F1 micro-controller. To speedup the evaluation phase, we first used MAPS simulator to reduce the majority of leakages. Then, we addressed remaining leakages with our side-channel workbench. The fact that masked gadgets were fully written in assembly greatly improved the detection of leaky operations.

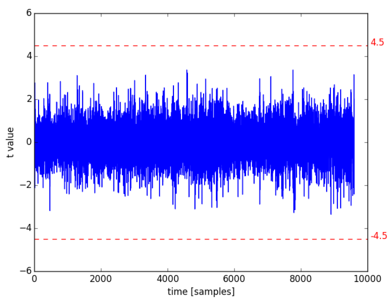
We only evaluated the most optimized masking of Dilithium, i.e. with power of two arithmetic since the prime version performances are prohibitive. The T-test evaluation of arith::to::bool::lowbits, arith::to::bool::highbits, arith::makehint and arith::rejection after 10,000 traces are provided Figure 6.



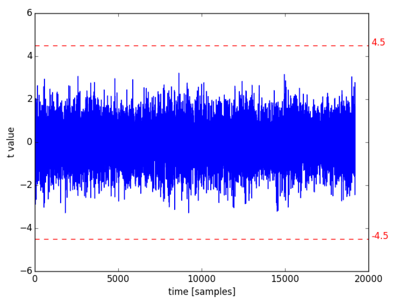
(a) bool::rejection



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



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



(d) arith::makehint

Fig. 6: Evaluation of the T-test on masked gadgets after 10.000 traces.

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.

7 Conclusion

In this paper, we described how to efficiently mask the Dilithium signature scheme. Our approach is based on a slight modification of the reference implementation of Dilithium by setting a power of two modulus instead of prime.

This optimization greatly reduces the complexity of decomposition operations such as $\mathrm{LowBits}_q$ or $\mathrm{HighBits}_q$, reducing computation times by a factor up to 8. Regarding the overhead compared to a non-masked implementation, the order-1 masking is slower by approximately a factor of 5.6, 11.6 for order-2 masking and 28 for order-3 masking.

We also provided a side-channel leakage analysis for both non-masked and masked of version of Dilithium on STM32F1 micro-controller. We were able to successfully found some leakages on decomposition functions and the rejection operation after no more than 500 traces for the non-masked version while our protected implementation did not show first-order leakage for 10.000 traces.

Acknowledgment

We acknowledge the support of the french Programme "Investissement d'Avenir" under the national project RISQ P141580-2660001 / DOS0044216.

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: Proc. of EUROCRYPT (2018)
4. Bindel, N., Buchmann, J., Krämer, J.: Lattice-Based Signature Schemes and Their Sensitivity to Fault Attacks. In: Proc. of FDTC (2016)
5. Chari, S., Jutla, C.S., Rao, J.R., Rohatgi, P.: Towards Sound Approaches to Counteract Power-Analysis Attacks. In: Wiener, M.J. (ed.) Proc. of CRYPTO. Lecture Notes in Computer Science, vol. 1666, pp. 398–412. Springer (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: Proc. of COSADE. pp. 82–98 (2018)
7. Duc, A., Dziembowski, S., Faust, S.: Unifying Leakage Models: From Probing Attacks to Noisy Leakage. In: Advances in Cryptology – EUROCRYPT 2014 (2014)
8. Ducas, L., Kiltz, E., Lepoint, T., Lyubashevsky, V., Seiler, G., Stehlé, D.: CRYSTALS-DILITHIUM, Algorithm Specifications and Supporting Documentation. https://pq-crystals.org/dilithium/data/dilithium-specification.pdf (2017)
9. Ducas, L., Lepoint, T., Lyubashevsky, V., Schwabe, P., Seiler, G., Stehlé, D.: CRYSTALS - Dilithium: Digital Signatures from Module Lattices. Cryptology ePrint Archive, Report 2017/633 (2017)

10. Espitau, T., Fouque, P.A., Gérard, B., Tibouchi, M.: Loop-Abort Faults on Lattice-Based Fiat-Shamir and Hash-and-Sign Signatures. In: Proc. of SAC 2016 (2017)
11. Espitau, T., Fouque, P., Gérard, B., Tibouchi, M.: Side-channel attacks on BLISS lattice-based signatures: Exploiting branch tracing against strongswan and electromagnetic emanations in microcontrollers. In: Proc. of CCS. pp. 1857–1874 (2017)
12. Groot Bruinderink, L., Hülsing, A., Lange, T., Yarom, Y.: Flush, Gauss, and Reload – A Cache Attack on the BLISS Lattice-Based Signature Scheme. In: Cryptographic Hardware and Embedded Systems – CHES 2016 (2016)
13. Ishai, Y., Sahai, A., Wagner, D.A.: Private Circuits: Securing Hardware against Probing Attacks. In: Proc. of CRYPTO. pp. 463–481 (2003)
14. 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), http://eprint.iacr.org/2016/517
15. Pessl, P., Groot Bruinderink, L., Yarom, Y.: To BLISS-B or Not to Be—Attacking Strongswan's Implementation of Post-Quantum Signatures. In: Proc. of Computer and Communications Security (CCS) (2017)
16. Prouff, E., Rivain, M.: Masking Against Side-Channel Attacks: A Formal Security Proof. In: Advances in Cryptology – EUROCRYPT 2013 (2013)
17. Regev, O.: On Lattices, Learning With Errors, Random Linear Codes, and Cryptography. In: Proc. of STOC (2005)

A Decompose$_q$ for prime module

Algorithm 4 Decompose(r).

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

```
$r_0 = [r]_\alpha$
 1: $t_0$      $= r \wedge \text{0x7FFFF}$
 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 $(r_0, r_1)$
```

B Common masked gadgets

Algorithm 5 bool::maskfromsign($(r)_{0 \le i < t}$). Gadget that computes a mask from the sign of a boolean masked shares. ω is the computer word base (usually 32 or 64)

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

Algorithm 6 arith::to::bool::convert($(a)_{0\leq i<t}$)

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

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

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

Algorithm 8 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 $(r)_{0\leq i<t}$
```

C Gadgets for Dilithium with prime modulus

Algorithm 9 arith:makeint($(r)_{0\leq i<t}, (z)_{0\leq i<t}, \beta$). Masked algorithm of MakeHint$_q$ with a prime mudulus q. w is the word base (usually 32 or 64).

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

Algorithm 10 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($q-\beta-1$)
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 11 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 $(r)_{0\leq i<t}$
```

Algorithm 12 Decompose(r) with constant time implementation, for prime modulus $q = 8380417$ and $\alpha = 523776$

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

D Gadgets for Dilithium with power of two modulus

Algorithm 13 arith::generate(β). Generates a uniformly sampled integer in the bounds $[-\beta, +\beta]$ for modulus $q = 2^b$. ω is the computer word size (usually 32 bits or 64 bits).

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

Algorithm 14 arith::to::bool::lowbits($(r)_{0 \leq i < t}, \beta$). Masked implementation of LowBits_q with modulus $q = 2^b$. ω is the computer word base (usually 32 bits or 64 bits).

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

Algorithm 15 arith::to::bool::highbits($(r)_{0\leq i<t}, \beta$). Masked implementation of $\mathrm{HighBits}_q$ with modulus $q = 2^b$. ω is the computer word base (usually 32 bits or 64 bits).

```
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 16 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 17 arith:makeint($(r)_{0\leq i<t}, (z)_{0\leq i<t}, \beta$). Masked algorithm of $\mathrm{MakeHint}_q$ with modulus $q = 2^b$. ω is the computer word base (usually 32 bits or 64 bits).

```
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 $c$
```