Leveling Dilithium against Leakage

Revisited Sensitivity Analysis and Improved Implementations

Melissa Azouaoui[1], Olivier Bronchain[1,2], Gaëtan Cassiers[2,3,4], Clément Hoffmann[2], Yulia Kuzovkova[1], Joost Renes[1], Tobias Schneider[1], Markus Schönauer[1], François-Xavier Standaert[2] and Christine van Vredendaal[1]

[1] NXP Semiconductors, firstname.lastname@nxp.com
[2] UCLouvain, Belgium, firstname.lastname@uclouvain.be
[3] Graz University of Technology, Austria, firstname.lastname@iaik.tugraz.at
[4] Lamarr Security Research, Austria

Abstract. CRYSTALS-Dilithium has been selected by the NIST as the new standard for post-quantum digital signatures. In this work, we revisit the side-channel countermeasures of Dilithium in three directions. First, we improve its sensitivity analysis by classifying intermediate computations according their physical security requirements. This allows us to identify which parts of Dilithium must be protected against Differential Power Analysis (DPA), which parts must be protected against Simple Power Analysis (SPA) and which parts can leak in an unbounded manner. Second, we provide improved gadgets dedicated to Dilithium, taking advantage of recent advances in masking conversion algorithms. Third, we combine these contributions with standard shuffling techniques in order to design so-called leveled implementations that offer an improved security vs. performance trade-off compared to the state-of-the-art. Our benchmarking results additionally put forward that the randomized version of Dilithium can lead to significantly more efficient implementations (than its deterministic version) when side-channel attacks are a concern.

Keywords: Dilithium · Masking · Lattice-based Cryptography · Post-Quantum Cryptography · Side-Channel Countermeasures

1 Introduction

The world's digital security infrastructure has always relied on a range of efficient and secure cryptographic primitives, including both symmetric and asymmetric solutions. In particular for asymmetric cryptography, RSA and ECC are the ubiquitous schemes in practice. However, with the advent of powerful and dedicated quantum computers, the established asymmetric cryptographic schemes, that we mainly use for key exchange and digital signatures, will no longer provide the desired security.

In 2016, the National Institute of Standards and Technology (NIST) has launched a standardization effort for cryptographic schemes that can withstand quantum cryptanalysis [Nat]. Recently in 2022, the NIST announced the first Post-Quantum Cryptography (PQC) schemes to be standardized. Mainly, (CRYSTALS-)Kyber [ABD+19] for Key Encapsulation Mechanism (KEM), and (CRYSTALS-)Dilithium [DLL+17] for digital signatures. Both Kyber and Dilithium are lattice-based schemes, and in recent years the analysis of lattice-based PQC schemes and their implementations has become a prominent area of research. This is not only due to their widely accepted strong security but also because of their implementation efficiency in comparison to other PQC schemes.

Although a PQC scheme can be secure against classic and quantum adversaries, this is not sufficient to provide practical security in the embedded context. The implementations

of cryptographic schemes on constrained devices can be targeted by physical attacks, which include Side-Channel Analysis (SCA) and Fault Injection (FI) attacks. Over the last years, PQC KEM's have attracted most of the attention when it comes to SCA. Indeed, most KEM's in the NIST competition, including Kyber, rely on the Fujisaki-Okamoto (FO) transformation [FO99] which is a simple and generic technique to achieve IND-CCA security. Unfortunately, the leakage of the re-encryption step in the FO transformation leads to very powerful SCA's, demonstrated and analyzed in many recent works, including but not limited to [RRCB20, REB+22, UXT+22]. An adversary can also exploit leakage from the Number Theoretic Transformation (NTT) or from the Key Derivation Function (KDF) to extract the long term secret key or the shared secret key [RPBC20, HHP+21, KPP20, PPM17]. This variety of threats implies a large attack surface leading to significant overheads when protecting PQC KEM's against SCA's [ABH+22].

To the best of our knowledge, digital signatures, including Dilithium, have received much less attention than KEM's with respect to SCA. The main results include a work by Ravi et al. [RJH+18] that shows that to achieve existential forgery an attacker only requires knowledge of one part of the secret key in Dilithium, namely $\mathbf{s_1}$. Marzougui et al. [MUTS22] exploit leakage of the zero coefficients in the secret signing nonce $\mathbf{y}$ for multiple signatures and recover the secret key by leveraging least squares regression and integer linear programming. Liu et al. [LZS+21] also present an SCA on Dilithium, which is able to recover the secret key from the leakage of a single bit of the secret signing nonce $\mathbf{y}$ for multiple signatures. The authors use this side-channel information to define a problem called the Fiat-Shamir Integer LWE, and show that it can be solved efficiently. This attack is very reminiscent of the well-known lattice reduction attacks on (EC)DSA (and other Schnorr-like signature schemes) with partial nonce leakage, originally due to Howgrave-Graham and Smart [HS01] and very recently improved by Sun et al. [SETA22]. Liu et al. showed that their attack requires a relatively low number of signatures. This result, along with previous works and the fact that the side-channel analysis of Dilithium is quite a new research topic for the community, highlights the vital need to protect the future digital signature standard against these threats and future threats.

The amount of published works appears to be even scarcer when it comes to protecting Dilithium against side-channel attackers. Again to the best of our knowledge, the main contribution comes from Migliore et al. [MGTF19]. They present masked gadgets for Dilithium, and a power-of-two modulus masked version of it.

Contributions. In this work we tackle the challenge of efficiently protecting Dilithium implementations on embedded devices. Our contributions are the following.

First, we revisit the sensitivity analysis of Migliore et al. [MGTF19]. Interestingly, we notice that the authors do not consider some intermediates as sensitive even though they can be explicitly used to recover the secret key. Conversely, some were unnecessarily protected since they could be computed from the signature and the public key. These observations lead to improved security and to more efficient signature generation.

Second, based on our sensitivity analysis we propose new and improved masked gadgets for the main operations of Dilithium (namely the bound check, the secret sampling and the decomposition) and for all NIST security levels. To the best of our knowledge, our work presents the first masked Dilithium design compliant with the future standard for all the parameter sets. Then, inspired by the analysis of leveled implementations by Azouaoui et al. [ABH+22], we additionally discuss how to level the implementation of Dilithium by identifying SPA and DPA targets for both its deterministic and randomized versions, and preventing these attack vectors with the appropriate (masking and shuffling) countermeasures. In other words, we use our revised sensitivity analysis to describe efficient strategies to protect ephemeral and long term secrets of Dilithium against leakage.

Finally, we provide a complete benchmark for an ARM Cortex-M4 microcontroller,

which includes individual components, their comparison with Migliore et al., and performances of full signature generation for deterministic and randomized versions of Dilithium. Our results show that significant performance improvements can be achieved by leveling the implementation. Furthermore, they highlight the advantages of randomized Dilithium compared to its deterministic variant in the context of physical attacks.

2 Background

We next detail the notations used in the paper and the Dilithium signature scheme.

2.1 Polynomial arithmetic notations

All arithmetic operations in the paper are denoted over the polynomial ring $R = \mathbb{Z}_q[X]/(X^n + 1)$. We denote a polynomial with small caps such as $p \in R$, a vector of polynomials with bold letters such as $\mathbf{x} \in R^k$ and a matrix of polynomials with capital bold letters such as $\mathbf{X} \in R^{k \times k'}$. For Dilithium, the parameters of the ring are the prime $q = 2^{23} - 2^{13} + 1$ and the degree $n = 256$. For $z, \alpha \in \mathbb{Z}$ we write $z \bmod^{\pm} \alpha$ to mean the unique integer z' in $]-\frac{\alpha}{2}, \frac{\alpha}{2}]$ (resp., $[-\frac{\alpha-1}{2}, \frac{\alpha-1}{2}]$) with $z \equiv z' \mod \alpha$ if α is even (resp., odd). The notation $\mathbf{z} \bmod^{\pm} \alpha$ implies that all the coefficients in $\mathbf{z}$ are given with $\bmod^{\pm} \alpha$. With this, we can define the following norms on $\mathbb{Z}_q$, R and R^k respectively:

$$\|z\|_\infty = |z \bmod^{\pm} q|, \qquad \|p\|_\infty = \max_i \|p_i\|_\infty, \qquad \|\mathbf{w}\|_\infty = \max_i \|\mathbf{w}_i\|_\infty,$$

with $z \in \mathbb{Z}_q$, $p \in R$, p_i being the i-th coefficient of p, $\mathbf{w} \in R^k$ and $\mathbf{w}_i$ being the i-th polynomial in $\mathbf{w}$. Additionally, we define $S_\eta = \{w \in R : \|w\|_\infty \leq \eta\}$ and $\tilde{S}_\eta = \{w \bmod^{\pm} 2\eta : w \in R\}$. This means that the coefficients of an element in S_η or $\tilde{S}_\eta$ are in the range $[-\eta, \eta]$ or $]-\eta, \eta]$, respectively. We use the notation $x \leftarrow \mathcal{X}$ whenever we assign a uniformly random element of a set $\mathcal{X}$ to a variable x. The symbol $\|$ is used for the concatenation of two bit strings, the function `H` is an expandable output function (`XOF`).

2.2 Dilithium

Dilithium is a digital signature scheme based on the MLWE (Module Learning With Errors) and the SelfTargetMSIS (Module Short Integer Solution) problems [LS15]. It is the primary standard selected by the NIST for quantum safe digital signatures. Its main features are: random sampling from a uniform distribution instead of a discrete Gaussian distribution, a focus on keeping the public key and the signature as small as possible in terms of their bit size, and being easy to adjust for different security levels by only changing the dimensions of the matrices and vectors involved. For a comprehensive description of the algorithm we refer to the proposal [DLL+17] and the supplementary material A. Note that the pseudocode presented there and in the rest of the paper is a variation from the reference implementation described in [DLL+17, p.17]. The key differences are highlighted in Section 3.3. In this paper we refer to the implemented version if not stated otherwise. We describe the Dilithium key generation and signature generation algorithms in the following paragraphs. Since the verification does not involve long-term secret variables (and therefore does not leak sensitive information), we do not consider it in this work. Table 1 provides the Dilithium parameters for different NIST security levels.

3 Sensitivity analysis

In this section, we analyze Dilithium's key generation and signature algorithms and discuss the sensitivity of all the variables and functions potentially leading to side-channel attacks.

Table 1: Dilithium parameters.

NIST Security level	2	3	5
q (modulus)	$2^{23} - 2^{13} + 1$	$2^{23} - 2^{13} + 1$	$2^{23} - 2^{13} + 1$
d (number of dropped bits from $\mathbf{t}$)	13	13	13
τ (# of $\pm$ 1's in c)	39	49	60
γ_1 ($\mathbf{y}$ coefficient range)	2^{17}	2^{19}	2^{19}
γ_2 (low order rounding range)	(q-1)/88	(q-1)/32	(q-1)/32
(k, l) (dimensions of $\mathbf{A}$)	(4,4)	(6,5)	(8,7)
η (secret key range)	2	4	2
β ($= \tau \cdot \eta$)	78	196	120
ω (max. # 1's in $\mathbf{h}$)	80	55	75
average number of signing iterations	4.25	5.1	3.85

This sensitivity analysis indicates which operations/variables need to be protected against leakage. Following, we also explain how to level the implementation of Dilithium. That is, how to exploit the different sensitivity of different operations in order to protect them with the appropriate (more or less expensive) countermeasures. Doing so we also compare our analysis to the one previously proposed in [MGTF19].

3.1 Motivation & Methodology

Leveling was first proposed by Pereira, Standaert and Vivek [PSV15] and later applied to several symmetric authentication, encryption and authenticated encryption schemes (see for example Bellizia et al.'s survey [BBC+20]). The main observation shared by all these works is that the leakage-resistance of composite cryptographic constructions can be translated into minimum security requirements for its different components. In turn, these requirements can be fulfilled efficiently with different protection mechanisms resulting in more efficient side-channel protected implementations.

Classification. The first step to level an implementation is to classify the different types of attacks that can be performed on the intermediate variables. We next classify them into three categories as introduced in [BBC+20] and applied to Kyber in [ABH+22]. The first category is Differential Power Analysis (DPA), where the adversary can gain information on a secret thanks to her access to a large number of leakage traces with a varying input. This setting corresponds to the standard context when targeting (the first-round S-box of) a block cipher implementation. The second class of attacks is called Simple Power Analysis (SPA), for which the adversary can only observe leakages for a small (bounded) number of different inputs (possibly with averaging). This setting typically corresponds to the case of attacks against the ephemeral secrets manipulated in an ECC implementation with point randomization, or to the case of attacks against the key scheduling part of a block cipher implementation. A third category is for data that can be fully leaked to the adversary without compromising the security of the cryptographic primitive, either because it is public or because we can prove this leakage does not harm the security.

Generally, it is more expensive to protect a variable against DPA's than against SPA's, while values that can leak in an unbounded manner do not require any protection. In the following, we therefore represent Dilithium's algorithms as block diagrams, with different colors used for each category of attack. Namely red is used when DPA resistance is required, orange for SPA resistance and blue when the value can be fully leaked.

Type of countermeasures. The previously detailed categories imply that the different parts of a Dilithium implementation may need to resist DPA and SPA, raising the question of which countermeasure to implement in each case.

The mainstream solution to prevent DPA is masking, and our following investigations will therefore rely on this standard choice. Precisely, we will use arithmetic masking to protect polynomial operations, Boolean masking to protect hash function calls, and set the number of shares d as our main security parameter. We refer to Section 4 for additional details on masking as well as new gadgets dedicated to Dilithium.

By contrast, and as discussed in [BGS15, UBS21], masking is not the most cost-effective solution to protect ephemeral secrets. This is because the security gains that are obtained by bounding the number of observed leakages and by masking are only summed when both countermeasures are combined. By contrast, it was shown that parallelism in hardware and shuffling (which emulates parallelism) in software can lead to SPA security more efficiently. Interestingly, shuffling has already been studied in the context of lattice-based cryptography, especially for the `NTT`'s [RPBC20, HSST22]. In this work, we will leverage a shuffled implementation of the `NTT` from [RPBC20] to protect SPA targets. We refer to Subsection 5.2 for details about the instantiation of this SPA countermeasure.

Concretely our goal is therefore to combine the shuffling countermeasure with an appropriate amount of masking. As will be confirmed in Section 6, such a leveled approach leads to a better trade-off between the physical security and the performances of an implementation, since it allows selecting the appropriate countermeasures for each target computation and to combine countermeasures in a more cost-effective manner.

3.2 Application to Dilithium

In the following, we classify all the intermediate variables involved in Dilithium's key generation and signature generation algorithms as DPA targets, SPA targets and public values. To do so, we first classify variables between sensitive ones (for DPA or SPA) or public values. We then differentiate between DPA and SPA targets. The resulting classification is summarized in color-coded diagrams: Figure 1 and Figure 2.

Starting with generalities, we first note that the public key can be leaked to the adversary over the whole scheme (since it is public). The public matrix $\mathbf{A}$ can also be leaked since it is deterministically derived from ρ. A similar status holds for some parts of the secret key $sk := (\rho, K, tr, \mathbf{s}_1, \mathbf{s}_2, \mathbf{t}_0)$, since similar variables are contained in the public key. Concretely, tr does not need to be protected either since it is a hash of pk. We additionally note that the vector of polynomials $\mathbf{t}_0$ can be leaked as well. Indeed, the Dilithium security proofs consider $\mathbf{t}$ (hence $\mathbf{t}_0$ and $\mathbf{t}_1$) to be public [DLL+17].[1] Furthermore, we do not consider message recovery attacks on the message M. As a result, only the vector of polynomials $\mathbf{s}_1$ and $\mathbf{s}_2$, and K, must be protected against DPA in order to avoid side-channel attacks leading to a signature forgery. These variables are denoted in filled red circles in Figure 1 and Figure 2. Next, we detail which other variables must be protected against side-channel attacks to avoid the leakage of long-term sensitive secrets. We start with their sensitivity analysis for the key generation followed by the signing procedure.

3.2.1 Key generation sensitivity

During key generation, the variable ζ has to be protected as it is the seed for all subsequent values (e.g., K). Similarly, ς has to be protected as well, since it serves as a seed to deterministically generate the long term secrets $\mathbf{s}_1$ and $\mathbf{s}_2$. Since key generation is performed only once, an attacker only has access to a single trace, hence the orange coloring corresponding to SPA. As described above, all the other variables in the key generation can be leaked or are public, hence do not need side-channel protection.

[1] As a result $\mathbf{t}$ could be fully part of the public key. In order to reduce the size of the public key, $\mathbf{t}$ is decomposed into $\mathbf{t}_0$ and $\mathbf{t}_1$. The public key only contains $\mathbf{t}_1$ instead of $\mathbf{t}$. This choice reduces the size of the public key by a factor close to two at the cost of a slightly increased signature size. However, the secret key size increases since it must contain $\mathbf{t}_0$.

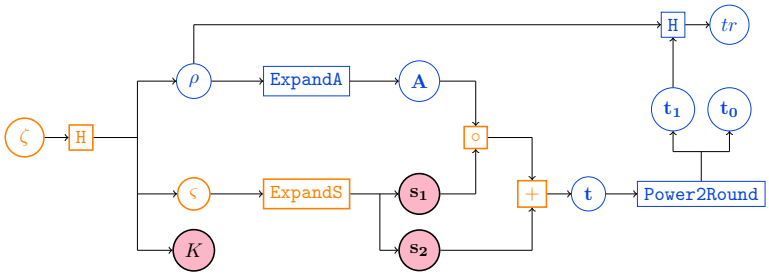


Figure 1: Graphical representation of the key generation. Output: $pk = (\rho, \mathbf{t_1})$, $sk = (\rho, K, tr, \mathbf{s_1}, \mathbf{s_2}, \mathbf{t_0})$. Red filled circles: long term secret keys (aka DPA targets). Orange: SPA protection required. Blue: no side-channel protection required.

3.2.2 Signature generation sensitivity

As detailed above, both tr and $\mathbf{t_0}$, the message M, the seed ρ, the hash μ and the public matrix $\mathbf{A}$ are public. However, the vector of polynomials $\mathbf{y}$ is sensitive and must be protected. Indeed, given a valid signature $\sigma = (\tilde{c}, \mathbf{z}, \mathbf{h})$, the secret vector $\mathbf{s}_1$ can be recovered from $\mathbf{z} = \mathbf{y} + c\mathbf{s}_1$ for known $\mathbf{y}$ [MUTS22]. A similar analysis applies to $\mathbf{w}_0$ which can lead to the recovery of $\mathbf{s}_2$. As a result, the vector of polynomials $\mathbf{w}$ must be protected: $\mathbf{w}_0$ is directly derived from $\mathbf{w}$, and it is possible to solve the system of equations $\mathbf{Ay} = \mathbf{w}$ for known $\mathbf{A}$ and $\mathbf{w}$ to recover $\mathbf{y}$ in most cases (see Section 3.3 for details). For the same reason, ρ' must be protected since it is used as a seed to obtain $\mathbf{y}$.

Continuing with the signing procedure, the vector of polynomials $\mathbf{w}_1$ is reconstructed from the signature and public key during verification. Hence, it cannot lead to an attack against the secret key.[2] Similarly, the challenge c can also be left unprotected since it is derived from $\mathbf{w}_1$ and public inputs. Then, before the bound check is performed, both $\mathbf{z}$ and $\tilde{\mathbf{r}}$ must be protected implying the need for secure bound check implementation. However, after successful rejection checks, they do not leak information about other sensitive values and can be leaked to the adversary.[3] For $\mathbf{z}$ this is trivial, as it is part of the signature. In the case of $\tilde{\mathbf{r}}$, this can be shown by the equation:

$$\mathbf{Az} - c\mathbf{t} = \mathbf{w} - c\mathbf{s}_2 = \alpha\mathbf{w}_1 + \tilde{\mathbf{r}}.$$

Indeed, for a valid signature, none of the values $\mathbf{A}$, $\mathbf{z}$, c, $\mathbf{t}$, and $\mathbf{w}_1$ are sensitive and α is a known parameter of the algorithm. Therefore, $\tilde{\mathbf{r}}$ can be computed using only public values, so there is no need to keep it protected after a successful signing process. A public $\tilde{\mathbf{r}}$ is quite handy, because it allows us to compute the hint $\mathbf{h}$ completely on public data.

Based on the previous classification, we finally need to determine the variables that are only SPA targets (meaning that the adversary can only observe a small number of traces with different inputs leading to information on those targets). For this purpose, we observe that the output of `ExpandMask` is indistinguishable from random since it is derived deterministically from the secret random seed ρ'. Therefore, the subsequent sensitive values (namely $\mathbf{y}, \mathbf{w}, \mathbf{w_0}$) are ephemeral, since they are obtained from the seed ρ' and fixed public matrix $\mathbf{A}$ without any other data allowing a DPA. It then remains that a SPA can be performed on `ExpandMask` based on the limited number of varying values for κ, which depends on whether the deterministic or randomized version of Dilithium is considered. (The same holds for $\mathbf{Ay}$ based on the fact that every polynomial in $\mathbf{y}$ is multiplied by

[2] The value $\mathbf{w}_1$ does not leak information about $\mathbf{y}$ after the signature is successfully calculated. This is due to the LWR (Learning With Rounding) assumption. The assumption is independent of rejection, and so $\mathbf{w}_1$ can be unmasked even when the corresponding signature gets rejected. See also [BG14].

[3] This refers to the rejection checks on $\mathbf{z}$ and $\tilde{\mathbf{r}}$. The one on $\mathbf{h}$ is not sensitive.

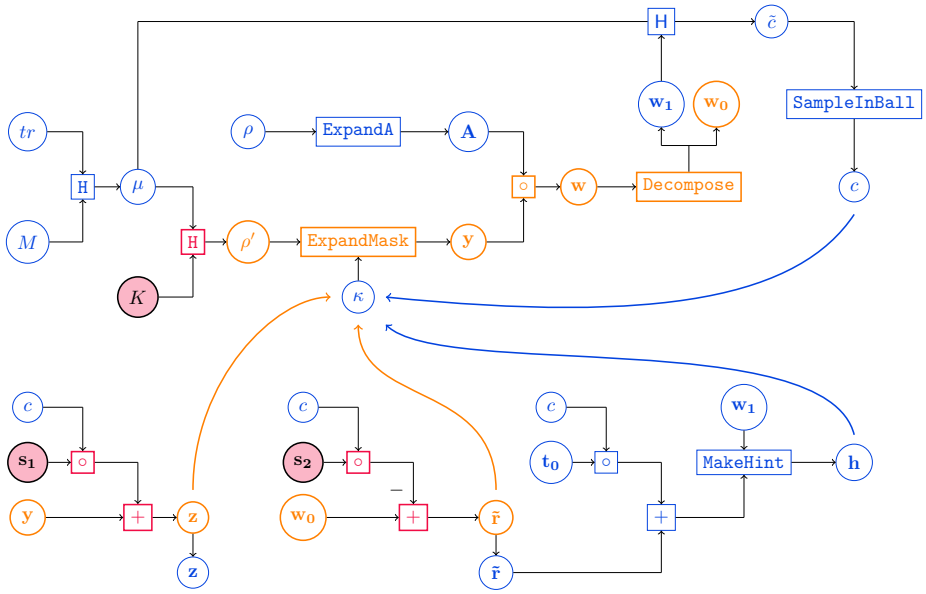


Figure 2: Graphical representation of the signing procedure, taking as input sk, M and outputting $\sigma = (\tilde{c}, \mathbf{z}, \mathbf{h})$. Curved arrows represent rejection checks. Red filled circles: long term secret keys (aka DPA targets). Red: DPA protection required. Orange: SPA protection required. Blue: no side-channel protection required.

multiple polynomials in $\mathbf{A}$.) In the deterministic case, the bound on the number of κ values that can be observed depends on the number of rejections and can reach ≈ 20 on average.[4] In the randomized case, it is tightly bound to 1. We emphasize that despite $\mathbf{y}$ and operations involving it are SPA targets, this value is – right after the secret key – the most sensitive variable in Dilithium signing. Hence, even when using more cost-effective SPA countermeasures only, it is important to ensure they offer sufficient security and Dilithium's randomized version comes with a quantitative benefit in this respect.

An intermediate attack path. The previous description suggests that the polynomial multiplication $c \cdot \mathbf{s}_1$ should be protected against DPA while $\mathbf{y}$ and $\mathbf{z}$ should be protected against SPA (a similar situation holds with $\mathbf{s}_2$, $\mathbf{w}_0$ and $\tilde{\mathbf{r}}$). This leads to the additional observation that leakage on the SPA-protected $\mathbf{y}$ and $\mathbf{z}$ could be used on order to gain information about the DPA-protected multiplication $c \cdot \mathbf{s}_1$. We next provide heuristic arguments why this attack path should not lead to simple attacks against Dilithium.

First, turning information on the (21-bit) $\mathbf{z}$ into information on the (8-bit) $c \cdot \mathbf{s}_1$ is affected by the algorithmic noise of the (20-bit) $\mathbf{y}$. For illustration, we show on Figure 3 the mutual information of the polynomial multiplication output given the Hamming weight of $\mathbf{y}$ and $\mathbf{z}$, with the bitsize of $\mathbf{y}$ used as a security parameter. As this bitsize increases, the mutual information decreases significantly below the mutual information of ≈ 2.5 that would be observed for an ephemeral leakage on $c \cdot \mathbf{s}_1$ without algorithmic noise.

Second, and as will be discussed next, $\mathbf{y}$ and $\mathbf{z}$ must be SPA-secure which in turn implies that their manipulation will be shuffled. Given the size of the polynomials in Dilithium, the shuffling will be over a 256-permutation which, if properly implemented, can further reduce the mutual information on $c\mathbf{s}_1$ by a factor 256 [VMKS12].

We conjecture the combination of these observations make this attack path difficult to

[4]For Level-3 parameters, $\mathbf{y}$ is sampled on average 5.1 times with `ExpandMask`. More precisely, within each `ExpandMask` execution, ρ' is hashed together with $l = 5$ independent inputs to generate all the polynomials in $\mathbf{y}$. This results on average in a total of 25.5 hashes of ρ' with independent inputs.

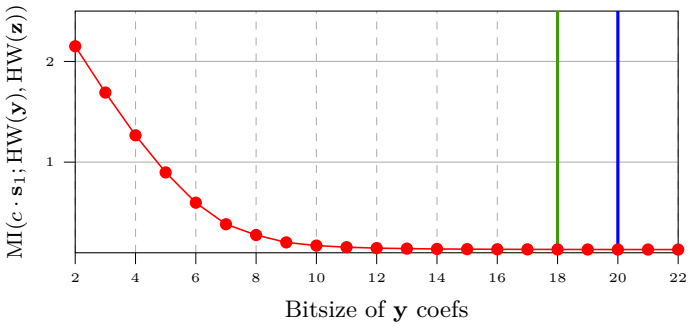


Figure 3: Mutual information leakage on $c \cdot \mathbf{s}_1$ given the Hamming weight of $\mathbf{y}$ and $\mathbf{z}$. Green (resp., blue) vertical line corresponds to Level-2 (resp., Level-3 and Level-5) parameters.

exploit in practice and leave its further investigation/formalization as an interesting scope for further research. In particular, we note that is shares both interesting similarities and significant differences with hard physical learning problems like Learning Parity with Leakage [DFH+16] (generalized to larger fields) or Learning With Physical Rounding [DMMS21] (tweaked with a polynomial structure and, more annoyingly, non-uniform secrets).

3.3 Differences with [MGTF19].

Most of our claims made above do align with the ones made in [MGTF19]. However, our conclusions on $\mathbf{w}$ and $\tilde{\mathbf{r}}$ slightly differ, which we discuss in the following.

Protecting $\mathbf{w}$. First we look at $\mathbf{w}$, in particular at the system of equations that produces it: $\mathbf{Ay} = \mathbf{w}$. It is possible to solve this system for $\mathbf{y}$, if the matrix $\mathbf{A}$ has one more row than columns. This is the case for NIST security levels 3 and 5, where $\mathbf{A}$ has dimensions 6×5 and 8×7 respectively. Even a simple solver is able to compute $\mathbf{y}$ in less than two minutes on a laptop, with original Dilithium parameters.[5] For level 2, since the matrix $\mathbf{A}$ is square (of dimensions 4×4) and random, it is most likely invertible.[6] Hence, with knowledge of $\mathbf{w}$, $\mathbf{y}$ can be computed simply as $\mathbf{y} = \mathbf{A}^{-1} \cdot \mathbf{w}$. This shows that $\mathbf{w}$ must be protected (or rather $\mathbf{w}_0$, since $\mathbf{w}_1$ is public), contrary to the approach in [MGTF19].

Unmasking $\tilde{\mathbf{r}}$. Before we look at $\tilde{\mathbf{r}}$ we need to address a variation of the signing procedure in Dilithium. The original pseudocode for the signing algorithm described in [DLL+17] only keeps the output $\mathbf{w}_1$ from `Decompose`($\mathbf{w}$). Then, instead of $\tilde{\mathbf{r}} = \mathbf{w}_0 - c\mathbf{s}_2$ it computes $\mathbf{r} = \mathbf{w} - c\mathbf{s}_2$. The rejection check on $\tilde{\mathbf{r}}$ is done on $\mathbf{r}_0$, which comes from `Decompose`($\mathbf{r}$). Also, the `MakeHint` function works slightly different and takes $\mathbf{r}, c, \mathbf{t}_0$ as input (but produces the same exact output $\mathbf{h}$). In [MGTF19] however, the $\mathbf{r}$-version is used while the $\tilde{\mathbf{r}}$-version is never mentioned. This is not a problem when it comes to comparing our security assessments. To see this, consider the equation:

$$\mathbf{r} = \mathbf{w} - c\mathbf{s}_2 = \alpha\mathbf{w}_1 + \mathbf{w}_0 - c\mathbf{s}_2 = \alpha\mathbf{w}_1 + \tilde{\mathbf{r}}.$$

[5] First, the polynomials of the first column of $\mathbf{A}$ are reduced to monic form via Gaussian elimination on the coefficients. Then, the fact that $a_{ij}X^k \cdot X^{n-k} = -a_{ij}X^0 (\text{mod } X^n + 1)$ is used to eliminate all but one polynomial in the first column. The remaining polynomial then allows us to eliminate all others in the first row, from column 2 onwards. The procedure is repeated on the second column, etc. The solver was implemented in Matlab and tested with the parameters given in [DLL+17] for security levels 3 and 5.

[6] Since the entries of $\mathbf{A}$ are elements of R, and R contains $q^n = 8380417^{256}$ elements, it is highly unlikely that two rows of $\mathbf{A}$ are linearly dependent. Therefore, the determinant of $\mathbf{A}$ is almost always non-zero and an inverse matrix exists.

This shows that $\mathbf{r}$ and $\tilde{\mathbf{r}}$ can be calculated from each other using the public values $\mathbf{w}_1$ and α. So any consideration regarding the sensitivity classification of one of these values, automatically applies to the other one as well. In [MGTF19], the value $\mathbf{r}$ is never unmasked which means that the calculation of the hint $\mathbf{h}$ must be protected against side-channel attacks. But as we explained above, $\tilde{\mathbf{r}}$ can be recreated from public values after a valid signature is output. So we consider $\tilde{\mathbf{r}}$ as public after the checks on $\mathbf{z}$ and $\tilde{\mathbf{r}}$.

4 Improved masked gadgets

In this section, we describe the techniques used for masking Dilithium. First, we recall some standard notions of masking along with the notations used in this paper. Then, we provide a set of new gadgets dedicated to Dilithium operations. For each of them, we justify their correctness and discuss their probing security. Finally, we discuss their instantiation to the case of the different parameter sets of Dilithium.

4.1 Masking background

Masking is a popular countermeasure against side-channel attacks. It consists in splitting any sensitive variable x into d shares [CJRR99]. Concretely, $d-1$ shares are chosen uniformly at random. Hence, any subset of $d-1$ shares remains independent of the secret x, forcing the adversary to exploit d shares simultaneously to extract sensitive information. This property must be maintained during the entire execution of the masked circuit. This is formalized in the probing model, ensuring that the adversary learns no information about the secret by having access to $d-1$ intermediate variables [ISW03].

In lattice-based cryptography, two types of masking are used. The first one is Boolean masking. In such a case, the sharing of a k-bit Boolean variable x is written as $\boldsymbol{x}^{B,k}$ and satisfies the property that $x = \bigoplus_{i=0}^{d-1} \boldsymbol{x}_i^{B,k}$ where $\boldsymbol{x}_i^{B,k}$ is the i-th share of x. The notation $\boldsymbol{x}^{B,k}[j]$ denotes the sharing of the j-th bit of x. Boolean masking is typically used for protecting symmetric primitives such as hash functions. The second one is arithmetic masking. In such a case, the sharing of a variable $x \in \mathbb{Z}_q$ is expressed as $\boldsymbol{x}^{A_q}$ such that $x = \sum_{i=0}^{d-1} \boldsymbol{x}_i^{A_q} \mod q$ where $\boldsymbol{x}_i^{A_q}$ is the i-th share of x. Arithmetic masking is typically used to perform polynomial operations such as additions and multiplications. Since both arithmetic and Boolean masking are used to protect lattice-based cryptography, gadgets are required to convert masking from one type to another. To convert from arithmetic to Boolean masking, we use $\texttt{SecA2BModp}_q^d$. Similarly to convert from Boolean masking to arithmetic masking, we use $\texttt{SecB2AModp}_q^d$. Eventually, we also leverage the gadget $\texttt{SecAddModp}_q^d$ that performs a modular addition operating on inputs protected with Boolean masking. We refer to [BC22] for an implementation of these algorithms.

In this work, the probing security is ensured thanks to the Probe Isolating Non Interference (PINI) security notion [CS20]. Fulfilling PINI ensures probing security and the composition of PINI gadgets is PINI as well. Since the new gadgets we propose can be expressed as composition of PINI gadgets previously proposed by Bronchain and Cassiers, it directly implies that they are PINI and therefore probing secure.

4.2 SecLeq

We first introduce $\texttt{SecLeq}_\psi^d\left(\boldsymbol{x}^{B,k}\right)$ described in Algorithm 2. It outputs a bit b equal to 1 if the input Boolean sharing of the k-bit variable x is less than or equal to a bound ψ.

Correctness. We next detail the correctness of Algorithm 2 for the case of $0 \leq \psi < 2^k - 1$. The first step in SecLeq consists in doing an addition of x with the $(k+1)$-bit two's

Algorithm 1 $\texttt{SecUnMask}_k^d\left(\boldsymbol{x}^{B,k}\right)$

Input: Boolean sharing $\boldsymbol{x}^{B,k}$ with $0 \leq x < 2^k$.
Output: Output the k-bit unmasked value x.

1: $\boldsymbol{y}^{B,k} \leftarrow \texttt{Refresh}_k^d\left(\boldsymbol{x}^{B,k}\right)$ ▷ Refresh based on the ISW multiplication [CS21, Algorithm 3].
2: $x \leftarrow \bigoplus_{i=0}^{d-1} \boldsymbol{x}_i^{B,k}$

Algorithm 2 $\texttt{SecLeq}_\psi^d\left(\boldsymbol{x}^{B,k}\right)$

Input: Boolean sharing $\boldsymbol{x}^{B,k}$ with $0 \leq x < 2^k$ and $\psi \geq 0$.
Output: For $0 \leq \psi < 2^k - 1$, public bit b with $b = 1$ if $x \leq \psi$ and $b = 0$ otherwise. If $\psi \geq 2^k - 1$, trivially returns $b = 1$.

1: $\boldsymbol{x'}^{B,k+1} \leftarrow \texttt{SecAdd}_{k+1}^d\left(\boldsymbol{x}^{B,k}, 2^{k+1} - \psi - 1\right)$
2: $b \leftarrow \texttt{SecUnMask}_1^d\left(\boldsymbol{x'}^{B,k+1}[k]\right)$

complement representation of $-(\psi+1)$ to obtain $x' = x - \psi - 1$. As a result, the output b must be set to 1 only if x' is strictly negative. Because of the input conditions $0 \leq x < 2^k$ and $0 \leq \psi < 2^k - 1$, the resulting x' is included in $-2^k \leq x < 2^k$ which fits in a $k+1$-bit two complement representation, hence no overflow occurs in the subtraction. The second step consists in unmasking the $(k+1)$-th bit of x' which corresponds to the sign bit of the two's complement representation. Eventually, the case of $\psi \geq 2^k - 1$ is trivial. Indeed, $x \leq 2^k - 1$ hence x is always smaller or equal to ψ.

Proposition 1. *Algorithm 2 is PINI if b is public.*

Proof. `SecUnMask` is PINI as a consequence of [CGMZ21, Lemma 2][7]. Therefore, if b is public, Algorithm 2 is a composition of PINI gadgets. □

Usage in Dilithium. `SecLeq` is not a high-level component of Dilithium but is instead used as a building block in Algorithm 3 and Algorithm 4. We note that the $\texttt{SecAdd}_{k+1}^d$ in `SecLeq` can be generically implemented with the full-adder based addition proposed in [BC22, Algorithm 6]. In the context of Dilithium, the added constant is public and fixed by the parameter set, enabling possible optimization of the adder taking into account the bits of the constant as well as the fact that only the sign bit (and so all the intermediate carries) must be explicitly computed. These optimizations depend on the constant and can lead to the saving of multiple `SecAnd`'s and XOR's.

4.3 `SecBoundCheck`

Algorithm 3 describes $\texttt{SecBoundCheck}_{q,\lambda_0,\lambda_1}^d\left(\boldsymbol{x}^{A_q}\right)$ which returns a bit b if the input arithmetic sharing $\boldsymbol{x}^{A_q}$ satisfies the property $-\lambda_0 \leq x \leq \lambda_1 \mod q$.

Correctness The first step in Algorithm 3 is to add λ_0 to the input sharing of x resulting in a sharing of x'. As a result, the output bit b will be set to one if and only if $0 \leq x' \leq \lambda_0 + \lambda_1 \mod q$. The second step is to check that condition thanks to `SecLeq`. To do so, the arithmetic sharing $\boldsymbol{x'}^{A_q}$ of x' is converted to a Boolean sharing $\boldsymbol{x'}^{B,k}$ thanks to `SecA2BModp`. The resulting sharing fulfills the input conditions of `SecLeq`. Indeed, $(x' \mod q) < q$ and $q < 2^k$ implies $x' < 2^k$. Additionally, since λ_0 and λ_1 are positive integers, we ensure that $\lambda_0 + \lambda_1 \geq 0$. The returned bit by `SecBoundCheck` is the one returned by `SecLeq`.

[7] `Refresh` is $(d-1)$-free-SNI [CS21]. Therefore, all its outputs and any set of at most t probes inside the gadget can be simulated by knowing its output and t of its input shares

Algorithm 3 $\texttt{SecBoundCheck}^d_{q,\lambda_0,\lambda_1}\left(\boldsymbol{x}^{A_q}\right)$

Input: Arithmetic sharing $\boldsymbol{x}^{A_q}$, integer $q < 2^k$ and $\lambda_0 + \lambda_1 < q$ with $\lambda_0 \geq 0$ and $\lambda_1 \geq 0$.
Output: Bit b with $b = 1$ if $-\lambda_0 \leq x \leq \lambda_1 \mod q$, $b = 0$ otherwise.

1: $\boldsymbol{x'}_0^{A_q} \leftarrow \boldsymbol{x}_0^{A_q} + \lambda_0 \mod q$ ▷ $b = 1$ iff $0 \leq x' \leq \lambda_1 + \lambda_0 \mod q$
2: $\boldsymbol{x'}^{B,k} \leftarrow \texttt{SecA2BModp}^d_q\left(\boldsymbol{x'}^{A_q}\right)$
3: $b \leftarrow \texttt{SecLeq}^d_{\lambda_0+\lambda_1}\left(\boldsymbol{x'}^{B,k}\right)$

Proposition 2. *Algorithm 3 is PINI.*

Proof. The first addition is applied only on the first share hence PINI. $\texttt{SecA2BModp}^d_q$ is PINI by [BC22, Proposition 4], and Algorithm 2 is PINI by Proposition 1. Hence, Algorithm 3 is PINI since it is the composition of PINI gadgets. □

Usage in Dilithium. `SecBoundCheck` can be used to perform both rejection checks $\|\mathbf{z}\|_\infty < \gamma_1 - \beta$ and $\|\tilde{\mathbf{r}}\|_\infty < \gamma_2 - \beta$, where γ_1, γ_2 and β are defined by Dilithium specifications (see Table 1). In the first case, `SecBoundCheck` is instantiated with $\lambda_0 = \lambda_1 = \gamma_1 - \beta - 1$, where the -1 is due to the strict inequality in the norm check. Similarly, in the second case, `SecBoundCheck` is instantiated with $\lambda_0 = \lambda_1 = \gamma_2 - \beta - 1$.

4.4 `SecSampleModp`

Algorithm 4 describes `SecSampleModp` which samples uniformly x over the range $-\phi_0 \leq x \leq \phi_1 \mod p$ and outputs an arithmetic sharing when provided with a masked uniform randomness stream $(\boldsymbol{x}_0^{B,k}, \boldsymbol{x}_1^{B,k}, \dots)$.

Algorithm 4 $\texttt{SecSampleModp}^d_{q,\phi_0,\phi_1}\left(\boldsymbol{x}_0^{B,k}, \boldsymbol{x}_1^{B,k}, \dots\right)$

Input: Bounds ϕ_0 and ϕ_1 with $\phi_0 \geq 0$, $\phi_1 \geq 0$ and $\phi_0 + \phi_1 < q$.
Output: Arithmetic sharing $\boldsymbol{x}^{A_q}$ with uniformly distributed x such that $-\phi_0 \leq x \leq \phi_1 \mod q$.

1: $k \leftarrow \lceil \log_2(\phi_0 + \phi_1 + 1) \rceil$
2: $i \leftarrow 0$
3: **while** $\neg\texttt{SecLeq}^d_{\phi_0+\phi_1}\left(\boldsymbol{x}_i^{B,k}\right)$ **do**
4: $\quad i \leftarrow i + 1$
5: $\boldsymbol{x}^{A_q} \leftarrow \texttt{SecB2AModp}^d_q\left(\boldsymbol{x}_i^{B,k}\right)$
6: $\boldsymbol{x}^{A_q}[0] \leftarrow \boldsymbol{x}^{A_q}[0] - \phi_0 \mod q$

Correctness. We first note that the output sharing should be uniform on a continuous range $[-\phi_0, \phi_1]$ which contains $\phi_0 + \phi_1 + 1$ integers. This range can be represented with k-bits such that $\phi_0 + \phi_1 + 1 \leq 2^k$. The first step in Algorithm 4 is to consume its uniformly distributed input bits under the form of sharing $\boldsymbol{x}_i^{B,k}$ while the obtained x is strictly larger than $\phi_0 + \phi_1$. This inequality is checked by leveraging `SecLeq` described in Algorithm 2. Once the inequality is not satisfied, the obtained x is uniformly distributed on the range $0 \leq x \leq \phi_0 + \phi_1$. The Boolean sharing of x is then converted into an arithmetic sharing $\boldsymbol{x}^{A_q}$. Finally, $-\phi_0 \mod q$ is added to this arithmetic sharing, resulting in x being uniformly distributed over $-\phi_0 \leq x \leq \phi_1 \mod q$.

Proposition 3. *Algorithm 4 is PINI assuming that whether each x_i satisfies $x_i > \phi_0 + \phi_1$ is public information and that $x_{i^*} \leq \phi_0 + \phi_1$ for some integer i^*.*

Proof. The assumptions imply that the output value of the `SecLeq` gadget calls are public and that the gadget terminates with the number of iterations being public. Therefore, the gadget `SecSampleModp` can be viewed as a circuit composed of PINI gadgets, hence Algorithm 4 is itself PINI. □

Usage in Dilithium. `SecSampleModp` is used during both key generation and signing Dilithium procedures. First, during `ExpandS` in key generation, a secret key coefficient x in $\mathbf{s_1}$ or $\mathbf{s_2}$ is sampled such that $-\eta \leq x \leq \eta$ where $\eta \in \{2, 4\}$ depending on the Dilithium parameter set. This sampling can be masked with $\texttt{SecSampleModp}^d_{q,\eta,\eta}(\cdot)$. For these parameters, rejections can occur and a fresh x passes the `SecLeq` check with probability $\frac{5}{8}$ and $\frac{9}{16}$, respectively. Second, during `ExpandMask` signature generation, a coefficient x of $\mathbf{y}$ is sampled such that $-\gamma_1 < x \leq \gamma_1$ where γ_1 is a power of two such that $\gamma_1 \in \{2^{17}, 2^{19}\}$ depending on the parameter set. As a result, this sampling can be masked thanks to $\texttt{SecSampleModp}^d_{q,\gamma_1-1,\gamma_1}(\cdot)$. For these parameters, no resampling of x is required. Indeed, $\texttt{SecLeq}^d_{2\gamma_1-1}(\cdot)$ is used which satisfies the trivial condition $\phi \geq 2^k - 1$ since $\phi = 2\gamma_1 - 1$. The k must not be evaluated at run time since it is directly derived from the Dilithium parameter set. As an example for the `ExpandMask` execution during signature generation, $k \in \{18, 20\}$ depending on the parameter set. We note that in both `ExpandS` and `ExpandMask`, the x is sampled from the output of a hash function, which is most efficiently protected using Boolean masking. This explains why we consider only sampling in Boolean domain.[8] Moreover, whether the samples have to be rejected is public information in the original security proof of Dilithium.

4.5 `SecDecompose`

The `SecDecompose` gadget presented in Algorithm 5 enables to compute the decomposition (w_1, w_0) of a coefficient w such that $w = \alpha w_1 + w_0 \mod q$ with $w_0 = w \bmod^{\pm} \alpha$. Concretely, we leverage the fact that w_1 can be leaked to the adversary since it is computed during signature verification, and hence must not be protected against side-channel attacks. The first step of our gadget is to derive w_1 from ${\boldsymbol{w}}^{A_q}$. Then, ${\boldsymbol{w_0}}^{A_q}$ is obtained by computing ${\boldsymbol{w_0}}^{A_q} = {\boldsymbol{w}}^{A_q} - \alpha \cdot w_1 \mod q$. To the best of our knowledge, there is no generic and efficient method for a masked division to compute ${\boldsymbol{w_0}}^{A_q}$ divided by α to get w_1. Hence, we next specialize the extraction of w_1 to the different parameter sets of Dilithium.

Correctness Level 2. For the NIST level 2 parameters of Dilithium, we have $\alpha = (q-1)/44$. Hence, $\alpha^{-1} = -44 \mod q$. For these parameters, w_1 can be extracted by performing a division with reminder such that:

$$\lfloor \frac{\alpha w_1 + w_0}{\alpha} \rceil = \lfloor (\alpha w_1 + w_0) \cdot \frac{-\alpha^{-1}}{q-1} \rceil, \tag{1}$$

$$\approx \lfloor (\alpha w_1 + w_0) \frac{-\alpha^{-1}}{q} \rceil, \tag{2}$$

which in turn can be performed by using the `Compress` function defined as:

$$\texttt{Compress}(x, \delta, q) = \lfloor \frac{x \cdot \delta}{q} \rceil \mod \delta, \tag{3}$$

[8] An alternative solution is to leverage arithmetic to arithmetic conversion. It would require to first generate a random sharing ${\boldsymbol{x}}^{A_{\phi_0+\phi_1}}$ resulting in a uniform x modulus $\phi_0 + \phi_1$. Then, that sharing of x must be converted to arithmetic sharing with modulus q such as ${\boldsymbol{x}}^{A_q}$. Yet to our knowledge, there exists no arithmetic to arithmetic conversion more efficient than the combination of `SecLeq` and `SecB2AModp`.

Algorithm 5 $\texttt{SecDecompose}^{d}_{q,\alpha}\left(\boldsymbol{w}^{A_q}\right)$

Input: Arithmetic sharing $\boldsymbol{w}^{A_q}$, prime integer q with $q < 2^k$ integer α with $0 < \alpha < q$ and $\alpha = 2\gamma_2$.
Output: Arithmetic sharing ${\boldsymbol{w_0}}^{A_q}$ and integer w_1 such that $w = \alpha w_1 + w_0 \mod q$.

1: **if** NIST Level 3 or Level 5 **then**
2: $\quad {\boldsymbol{b}}^{A_q} \leftarrow {\boldsymbol{w}}^{A_q} + \gamma_2 \mod q$
3: $\quad {\boldsymbol{b'}}^{A_q} \leftarrow \alpha^{-1} \cdot {\boldsymbol{b}}^{A_q} - 1 \mod q$
4: $\quad {\boldsymbol{b'}}^{B,k} \leftarrow \texttt{SecA2BModp}^{d}_{q}\left({\boldsymbol{b'}}^{A_q}\right)$
5: $\quad {\boldsymbol{w_1}}^{B,k'} \leftarrow {\boldsymbol{b'}}^{B,k}[[0, k'[]$
6: **else** ▷ NIST Level 2
7: $\quad {\boldsymbol{w_1}}^{B,k'} \leftarrow \texttt{SecCompress}^{d}_{q,-\alpha^{-1}}\left(\boldsymbol{w}^{A_q}\right)$
8: $w_1 \leftarrow \texttt{SecUnMask}^{d}_{k'}\left({\boldsymbol{w_1}}^{B,k'}\right)$
9: ${\boldsymbol{w_0}}^{A_q} \leftarrow {\boldsymbol{w}}^{A_p} - \alpha \cdot w_1 \mod q$

for which a masked version at any order is presented in [CGMZ21]. The `Compress` function can be used since $-\alpha^{-1} \ll q$. Hence, the error does not have an impact on the results. This fact has been checked exhaustively for all possible values of $w \mod q$.[9]

Correctness Level 3 & Level 5. Next, we check the correctness of `SecDecompose` for NIST level 3 and level 5 parameters. In such cases, $\alpha = (q-1)/16$. Hence, $\alpha^{-1} = -16 \mod q$. The first steps in Algorithm 5 execute the following processing to w in order to derive b' such that;

$$b' = \alpha^{-1} \cdot ((\alpha w_1 + w_0) + \frac{\alpha}{2}) - 1 \mod q, \tag{4}$$

which can be alternatively expressed as:

$$b = w_1 + \alpha^{-1} \cdot (w_0 - \frac{\alpha}{2}) \mod q, \tag{5}$$

$$= w_1 + 16 \cdot (\frac{\alpha}{2} - w_0) \mod q. \tag{6}$$

There, we note that $\frac{\alpha}{2} - w_0$ is strictly positive thanks to the definition of `Decompose`. Indeed, it follows from $-\alpha/2 \leq w_0 \leq \alpha/2$. As a result, w_1 can simply be contained in the 4 LSBs of the binary representation of b'. This is done thanks to the combination of $\texttt{SecA2BModp}^{d}_{q}$ and just keeping the 4 LSBs of the output.[10]

Proposition 4. *Algorithm 5 is PINI if w_1 is public.*

Proof. If w_1 is public, Algorithm 5 is the composition of PINI gadgets hence it is PINI. □

5 Implementation

We now discuss the different designs we compare later in Section 6. We first detail both the DPA and SPA countermeasures we consider. Second, we describe the implementations, both fully masked and leveled, and both for the deterministic and the randomized version of Dilithium. We stress that having leveled and fully masked implementations enables us to measure the gain offered by leveling. All our results are based on modified versions of the Dilithium implementations provided by the PQM4 project [KRSS]. These are C implementations with optimized assembly for polynomial arithmetic and hash functions.

[9] [CGMZ21] only considers δ equal to a power of two. We take δ as an arbitrary positive number.
[10] We note that only the LSBs of the $\texttt{SecA2BModp}^{d}_{q}$ have to be explicitly computed. As a result, this can save several `SecAnd` when the $\texttt{SecA2BModp}^{d}_{q}$ from [BC22] is used.

5.1 DPA countermeasures

To protect against DPA, we make use of masking with the gadgets presented in Section 4 and the underlying masked additions and conversions gadgets such as `SecA2BModp`, `SecAdd`, `SecAddModp` or `SecB2AModp`. We rely on their state-of-the-art bitsliced implementations introduced by Bronchain and Cassiers, which offer (to the best of our knowledge) the best performances on Cortex-M4 [BC22]. Eventually, we use the same masked `Keccak` as the one provided in [BC22]. We additionally leverage arithmetic masking with q modulus for all the polynomial operations and then apply share-wise the optimized polynomial arithmetic from the PQM4 implementations. Interestingly, the smaller modulus q' approach for the `NTT` in $\mathbf{s_1} \circ c$ proposed in [AHKS22] could also be used. However, it requires an arithmetic to arithmetic masking conversion from q' to q in order to perform the addition with $\mathbf{y}$. We leave the study of such a trade-off for future works.

5.2 SPA countermeasures

We start with SPA countermeasure for polynomial arithmetic and the Keccak permutation.

SPA protected polynomial arithmetic. In order to protect polynomial arithmetic against SPA, we leverage shuffling [HOM06, VMKS12]. Concretely, we use the fact that operations must be performed on the 256 coefficients independently. Hence, we first generate a permutation in the range $[0, 256[$ and then apply the operation on each of the coefficients according to that permutation. This is used for polynomial additions, subtractions, base-multiplications as well as norm checks. Our implementation is a plain C implementation, hence it does not benefit from optimized assembly as the unprotected version does.

For shuffling the `NTT`'s, Ravi et al. [RPBC20] present different techniques leading to a trade-off between cost and security. Attacks against this shuffling countermeasure have been studied in [HSST22]. Since the total cost of the `NTT`, even shuffled, remains relatively small compared to the total cost of the full execution, this paper focuses on the most secure shuffling countermeasure proposed in [RPBC20], named the *Coarse-Full-Shuffled* strategy. We directly re-use the code provided along with [RPBC20] for this task.

SPA protected Keccak. A first option to protect Keccak against SPA would be to shuffle its operations. However, shuffling with a relatively small permutation does not necessarily bring the expected SPA security on low noise devices since the permutation can then be directly targeted by the adversary and sometimes recovered with a single trace [UBS21].[11] Since Keccak only allows to shuffle on 5 to 25 independent elements, relying on a shuffled Keccak to achieve SPA security is a risky option. As a result, we rather assume that a Keccak coprocessor is available on our target microcontrollers (as it will likely be the method of choice to deploy post-quantum cryptography in a foreseable feature). As discussed in [USS+20, BMPS21], the latter is in general a very efficient option to obtain SPA security (which then relies on the parallelism of the implementation). For our following benchmarks, this Keccak hardware accelerator is assumed to perform one permutation round every clock cycle for a total of 24 Cycles per Keccak permutation.

5.3 Deterministic Dilithium

Deterministic Fully Masked. For the fully masked deterministic Dilithium, we rely on the gadgets presented above as well as a masked Keccak in software. Concretely, we use a masked Keccak for $\mathtt{H}(K||\mu)$ and within `ExpandMask`. In `ExpandMask`, the randomness

[11] The smallest permutation considered for polynomial operations is 256, which is significantly larger than the permutations over 25 independent operations studied in [UBS21].

generation in SecSampleModp (see Algorithm 4 Line-3) is performed with a call to the secure XOF (based on Keccak). The multiplication $\mathbf{Ay}$ is performed on each of the shares of $\mathbf{y}$ independently by leveraging the optimized arithmetic operations in [KRSS]. For the $\mathbf{w}$ decomposition, we leverage the new gadget SecDecompose presented in Algorithm 5 with the appropriate parameters given in Subsection 4.5. Similarly, the protected rejections $\|\mathbf{z}\|_\infty < \gamma_1 - \beta$ and $\|\tilde{\mathbf{r}}\|_\infty < \gamma_2 - \beta$ are implemented thanks to SecBoundCheck presented in Algorithm 3. Eventually, similarly to SecLeq in Algorithm 3, we unmask the public signature $\mathbf{z}$ and $\tilde{\mathbf{r}}$ using the SecUnMask gadget, in order to maintain probing security.

Deterministic Leveled. For the leveled variant of Dilithium, the only usage of Keccak that needs to be secured against DPA is $\mathtt{H}(K||\mu)$. Hence, for this purpose, we use the masked software implementation. ExpandMask uses only the SPA protected version of Keccak (coprocessor). The multiplication $\mathbf{Ay}$ and the decomposition of $\mathbf{w}$ are performed in a shuffled manner. Eventually, we note that the masked addition in $\mathbf{s}_1 c + \mathbf{y}$ requires to only add $\mathbf{y}$ to the first share of $\mathbf{s}_1 c$. This addition is shuffled as well. Interestingly, we note that no refresh is required for the unmasking as long as it is performed by adding the shares of $\mathbf{s}_1 c$ into the first one. Indeed, by doing so, each produced intermediate is an ephemeral secret thanks the fact that it is an addition with the ephemeral secret $\mathbf{y}$.[12]

5.4 Randomized Dilithium

The randomized version of Dilithium enables a larger degree of freedom in the generation of the uniform vector of polynomial $\mathbf{y}$ compared to the deterministic version. Indeed, for the deterministic version, the randomness sampling in Algorithm 4 is based on secured XOF (Keccak) which can be a bottleneck in terms of performances. For the randomized version, the first option is to directly generate the vector $\mathbf{y}$ from a TRNG. We also note that the previously mentioned TRNG can be replaced by an unprotected Keccak in case one lacks a trusted randomness source. In both cases, the resulting implementation does not weaken the security of Dilithium as $\mathbf{y}$ remains uniform even though it is not following the specification of the Dilithium third round submission.

Randomized Fully Masked. For the fully masked randomized version of Dilithium, randomness sampling in SecSampleModp (Line-3) can be directly performed with the TRNG instead of the costly masked XOF as done for the deterministic version. This produces a Boolean sharing of the uniform $\mathbf{y}$. A subsequent SecB2AModp must be applied to obtain an arithmetic sharing of $\mathbf{y}$, as detailed in Algorithm 4. All the other operations are performed similarly to the fully masked deterministic version.

Randomized Leveled. Finally, for the randomized leveled implementation of Dilithium, we consider the generation of $\mathbf{y}$ performed directly from the TRNG. All the other operations are performed similarly to the leveled deterministic version.

6 Benchmarks

In this section, we report the performances of the Dilithium implementations described in Section 5. Precisely, we first describe the benchmarking setup used for this purpose. Second, we report the performance improvement provided by the new gadgets of Section 4 compared to the ones of [MGTF19]. Then, we describe the cost of each individual operation in Dilithium's signature generation (without considering the rejections). Based on this, we

[12] For completeness, we also report in Supplementary Material B the performances of Dilithium signature generation when a custom software shuffled Keccak is used within ExpandMask.

discuss the benefits of leveled Dilithium signature generation, and we compare the performance of both deterministic and randomized versions when side-channel countermeasures are required. Performances are given for Dilithium with Level-3 parameters (see Table 1), but the resulting discussion applies to all Dilithium security levels.

6.1 Benchmarking setup

In order to evaluate the execution time, we use a similar benchmarking setup as the one provided in [BC22], which itself is based on the `PQM4` benchmarking initiative for PQC signatures and KEMs [KRSS]. More precisely, the benchmarks are performed with the NUCLEO-L4R5ZI demonstration board. The cycle counts are measured thanks to the cycle-accurate counter `DWT_CCYCNT`. With the considered clock configuration, the TRNG of the microcontroller provides 32 fresh random bits every 53 Cycles. This TRNG is used as the randomness for the side-channel countermeasures (i.e., shuffling and masking) as well as for the `ExpandMask` in randomized versions of Dilithium.

6.2 Gadgets improvements

In the following, we compare the gadgets presented in Section 4 and the corresponding ones in [MGTF19] proposed by Migliore et al. and report the results in Figure 4. To enable a fair comparison, we implemented the gadgets as described in [MGTF19] by leveraging the PINI property and the bitslice gadgets from [BC22] for `SecAdd`, `SecAddModp`, `SecA2BModp` and `SecB2AModp`. We note that [MGTF19] uses a parameter w reflecting the bus width of the target CPU, which implies that operations might be performed on more bits than necessary. In our implementation, we do not use that parameter w as the operations are performed on the exact necessary number of bits as allowed by bitslicing. Similarly, [MGTF19] performed masked `SecAnd` with public values to isolate bits on secret variables. Individual bits are trivially isolated in our implementations thanks to bitslicing.

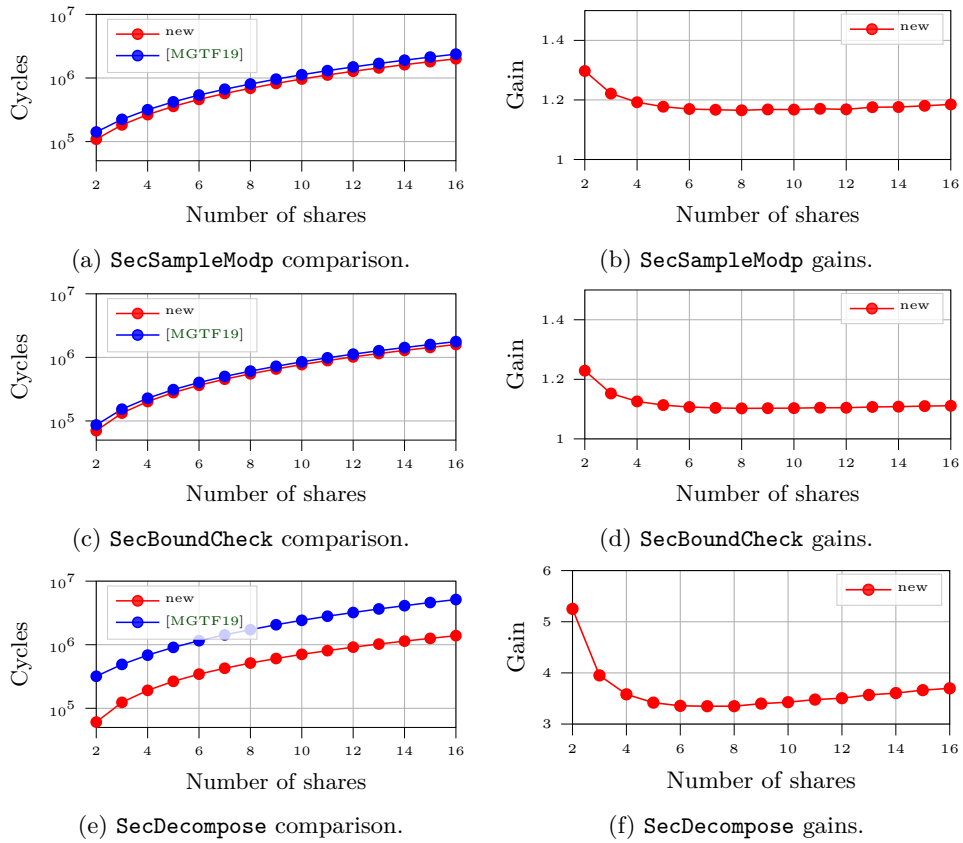
`SecSampleModp`. Both version of `SecSampleModp` as used in the signature generation are similar. The difference is that the subtraction with ϕ_0 is performed in Boolean masking for [MGTF19] and in arithmetic masking in Algorithm 4. As a result, our new gadget saves the cost of one `SecAdd` by replacing it by a share-wise addition. This results in a speedup of an approximate factor 1.2, as highlighted in Figure 4b.

`SecBoundCheck`. Our `SecBoundCheck` also simplifies the one proposed in [MGTF19] where a `SecA2BModp` is performed followed by two `SecAdd`'s.[13] Our construction replaces one of these additions by one arithmetically masked addition, which is almost free. This leads to a performance improvement by a factor ≈ 1.1, as reported in Figure 4d.

`SecDecompose`. Finally, we compare the two implementations of `SecDecompose`. Interestingly, the main improvement comes from the fact that we first extract $\mathbf{w}_1$ efficiently and then unmask it to compute $\mathbf{w}_0$. This improvement relieson the sensitivity detailed in Section 3.3: $\mathbf{w}_1$ is considered as sensitive by Migliore et al. while it is not in Algorithm 5. In short, the implementation based on [MGTF19] starts with a `SecA2BModp`, continues with several (≈ 10) additions and finally performs a `SecB2AModp` to obtain the arithmetic sharing of $\mathbf{w}_0$. The new gadget only requires a single `SecA2BModp` and some share-wise operations with arithmetic masking. Overall, the new gadget runs ≈ 3.8 times faster.

Finally, we note that for Level-2 parameters, the α changes and the gadget from Migliore et al. does not apply. Our implementation of `SecDecompose` for Level-2 parameters is

[13] Only `SecBoundCheck` is described for Boolean sharing in [MGTF19]. Here we assume that a `SecA2BModp` is performed before the end to match Algorithm 3 specifications.



(a) `SecSampleModp` comparison.

(b) `SecSampleModp` gains.

(c) `SecBoundCheck` comparison.

(d) `SecBoundCheck` gains.

(e) `SecDecompose` comparison.

(f) `SecDecompose` gains.

Figure 4: Comparison between new gadgets and [MGTF19] for NIST Level-3 parameters.

slightly slower than for Level-3 and Level-5. Indeed, in the `SecCompress`, the `SecA2BModp` must be performed on a slightly larger modulus increasing the cost by a factor ≈ 1.2.

6.3 Deterministic Dilithium Level-3 Components

The performance of each operation within the deterministic version of Dilithium is reported in Table 2. This table contains the numbers for both fully masked and leveled implementations. Next, we compare these two implementations. We observe that the signature generation is more efficient with the leveled implementation. For two shares, 24 986 kCycles are needed for the fully masked implementation vs. only 6757 kCycles for the leveled implementation. Hence, leveling offers an improvement by a factor ≈ 3.7. Similarly, for 8 shares, the leveled implementation is $\approx 17.58\times$ faster than the fully masked implementation. These gains can be explained by the following observations:

- **`ExpandMask`:** A large proportion of the gains provided by leveling are due to the cost difference between the `ExpandMask` of the two implementations. Indeed, the masked version requires the execution of a masked `XOF` followed by `SecSampleModp` (described in Algorithm 4). Hence, `ExpandMask` represents about 55 % of the total run time in the fully masked case. For the leveled implementation, `ExpandMask` is only required to be resistant to SPA, hence we leverage an unprotected HW coprocessor which significantly reduces the run time. Concretely both the expensive masked `XOF` and the `SecA2BModp` within Algorithm 4 are no longer required.

Table 2: Performance of the deterministic Dilithium Level-3 components: number of clock cycles when running on a `STM32L4R5` and using the TRNG for masking randomness (32-bit randomness every 53 Cycles). Reported numbers are in `kCycles`. The numbers are for a single execution of the component (does not consider repetitions due to rejections).

d	2		4		6		8	
	Masked	Leveled	Masked	Leveled	Masked	Leveled	Masked	Leveled
sign	24,986.8	6,757.8	70,708.4	8,100.7	131,252.0	9,641.8	201,737.5	11,471.5
`NTT`(s)	185.4	185.4	370.8	370.7	556.2	556.3	741.6	741.6
`ExpandA`	2,160.8	2,160.8	2,160.8	2,160.8	2,160.8	2,160.8	2,160.8	2,160.8
`H`$(K\|\|\mu)$	367.0	367.2	1,094.5	1,094.9	2,006.2	2,006.8	3,237.9	3,238.6
`ExpandMask`	12,860.2	3.0	37,913.5	3.0	69,920.0	3.0	110,770.7	3.0
$\mathbf{Ay}$	382.2	2,459.8	764.4	2,459.8	1,146.6	2,459.8	1,528.7	2,459.7
`Decompose`	2,971.8	134.7	9,443.7	134.8	18,475.8	134.8	27,724.9	134.8
$\mathbf{y} + \mathbf{s}_1 c$	174.9	238.3	349.8	387.4	524.5	536.5	699.5	685.5
$\mathbf{w}_0 - \mathbf{s}_2 c$	209.9	398.3	419.6	584.5	629.4	770.7	839.3	956.7
reject	4,826.7	216.3	16,522.9	216.2	33,062.4	216.2	50,462.1	216.2
umsk	372.8	119.2	1,174.7	195.3	2,258.1	285.4	3,041.5	344.6

- **`Decompose` and rejection checks:** A similar gain is obtained for `Decompose` and the rejections since the leveled implementation is based on shuffling instead of masking.

- **Polynomial arithmetic:** In this case, the most important observation is that the matrix multiplication $\mathbf{Ay}$ is cheaper in the fully masked implementation for the number of shares considered. Indeed, the multiplication is performed share-wise while it requires shuffling in the leveled version. Despite the complexity of the matrix multiplication increases linearly with the number of shares while it is constant for the shuffled version, the first solution is cheaper even up to 8 shares. A similar trend exists for $\mathbf{y} + \mathbf{s}_1 c$ and $\mathbf{w}_0 - \mathbf{s}_2 c$. In the fully masked case, the addition of $\mathbf{y}$ and $\mathbf{w_0}$ is performed independently on all the shares, whereas the leveled implementation only requires a single shuffled addition on the first share of $\mathbf{s}_i c$.

6.4 Randomized Dilithium Level-3 Components

The performance numbers for the randomized version of Dilithium are reported in Table 3. The trends are similar to the deterministic versions. For 8 shares, the leveled implementation is $\approx 14.1\times$ faster than the fully masked implementation. Compared to the results presented for the deterministic version in Table 2, the main difference lies in the `ExpandMask`'s cost. In both the leveled and fully masked cases, no `XOF` is used and the randomness is sampled from the TRNG. However, in the fully masked version, the TRNG is used to sample the Boolean shares in Algorithm 4 and a `SecB2AModp` is still required subsequently. Hence, `ExpandMask` remains an expensive block even in the randomized version of Dilithium, because its output must be masked. By contrast, for the leveled implementation, the TRNG is directly used to generate the uniform (unshared) $\mathbf{y}$, hence leading to an (almost) negligible cost. The other components (namely `Decompose`, rejection and polynomial operations) of the signature generation are shared between the deterministic and randomized version of Dilithium, therefore they show identical trends as discussed previously.

By comparing the numbers provided in Table 2 and Table 3, we observe that the randomized version of Dilithium is faster than the deterministic version in every scenario, which is mostly due to the dominating cost of the masked Keccak. Besides, we note that the randomized version of Dilithium also requires milder assumptions for its side-channel security to hold. First, it limits the SPA attack path discussed in Subsubsection 3.2.2 to lower complexities. More generally, it does not allow the adversary to average traces for the same inputs, which improves its security against SPA when the latter is based

Table 3: Performance of the randomized Dilithium Level-3 components: number of clock cycles when running on a `STM32L4R5` and using the TRNG for masking randomness (32-bit randomness every 53 cycles). Reported numbers are in `kCycles`. The numbers are for a single execution of the component (does not consider repetitions due to rejections).

d	2		4		6		8	
	Masked	Leveled	Masked	Leveled	Masked	Leveled	Masked	Leveled
sign	15,375.7	6,459.2	42,050.6	7,074.4	78,880.8	7,703.7	117,273.8	8,301.9
`NTT`(s)	185.4	185.4	370.7	370.7	556.2	556.2	741.7	741.6
`ExpandA`	2,160.8	2,160.7	2,160.8	2,160.7	2,160.7	2,160.7	2,160.7	2,160.7
`H`$(K\|\|\mu)$	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
`ExpandMask`	3,594.3	72.1	10,342.5	72.1	19,555.1	72.1	29,547.5	72.1
Ay	382.3	2,459.5	764.4	2,459.5	1,146.6	2,459.5	1,528.8	2,459.6
`Decompose`	2,977.9	134.6	9,445.6	134.7	18,475.4	134.7	27,724.6	134.7
$\mathbf{y} + \mathbf{s}_1 c$	174.9	238.2	349.7	387.2	524.6	536.3	699.5	685.4
$\mathbf{w}_0 - \mathbf{s}_2 c$	210.0	398.2	419.7	584.2	629.5	770.3	839.4	956.5
reject	4,841.9	216.1	16,528.4	216.1	33,062.2	216.1	50,457.0	216.1
umsk	372.7	119.2	1,174.5	195.2	2,257.7	285.3	3,041.3	344.5

on parallelism. The combination of these observations naturally calls for considering the randomized Dilithium in application contexts where side-channel attacks are a concern.

7 Conclusion and open problems

In this work, we analyzed side-channel protected implementations of Dilithium by mixing different contributions. First, we presented an updated sensitivity analysis for its key generation and signing algorithms. Our results show that a previous work in this direction was slightly flawed, with some parts leading to insecurities and other parts leading to inefficiencies. They also exhibit the interesting potential for leveling the implementations of Dilithium, by distinguishing between DPA and SPA targets. Based on our sensitivity analysis, we then presented new gadgets to mask Dilithium for all its parameter sets. Our gadgets improve over the state-of-the-art, leading to performance gains of factors up to of 3.8. They also fill gaps for which it was previously unknown how to efficiently apply masking (e.g., the decompose operation for the NIST Level 2 parameters). Finally, we presented the first masked and leveled implementations of Dilithium on an ARM Cortex-M4 microcontroller. Our benchmark confirms the interest of leveling the implementation, which can lead to significant performance improvements. Overall, our analysis and benchmark also highlight that the randomized variant of Dilithium provides notably better side-channel properties, thanks to a smaller attack surface allowing more efficient protected implementations. We therefore believe that it should be the default variant for embedded devices when side-channel leakage needs to be taken into account.

These results suggest two important directions for further research.

First, the sensitivity analysis on which we rely is based on the heuristic identification of SPA and DPA attack paths. While this approach provides an intuitive way to spot necessary conditions for secure implementations, it also leaves the formalization of sufficient conditions (with sound proofs in practically relevant leakage models) as a major challenge. In particular, and contrary to the context of symmetric cryptography where the granularity of leakage-resistant modes of operation is rather coarse [BBC+20], the implementation of post-quantum public-key algorithms typically relies on finer-grain blocks, which may therefore be more difficult to analyze. For example, the elementary operations in Figures 1 and 2 do not always correspond to cryptographic primitives with well-defined black box security guarantees. This challenge also leads to the more prospective question whether other (possibly new) designs could offer better security guarantees against leakage than Dilithium – for example by facilitating the reduction of the alternative attack path discussed

in Subsubsection 3.2.2 to hard physical learning problems.

Second, our investigations put forward the need to protect some computations against SPA and other computations against DPA. But here as well, the separation between these attacks is less strict than in the symmetric setting. For example, some of the SPA attack paths, despite indeed targeting ephemeral secrets, do not need to recover those secrets in full and may therefore be very powerful and hard to prevent (i.e., small leakages can be amplified thanks to mathematical cryptanalysis). As a result, another important open problem is to evaluate the proposals made in this paper for reaching SPA and DPA security concretely, in different implementation contexts, and to try improving them.

References

[ABD+19] Roberto Avanzi, Joppe Bos, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, John M Schanck, Peter Schwabe, Gregor Seiler, and Damien Stehlé. Crystals-kyber algorithm specifications and supporting documentation. *NIST PQC Round*, 3:4, 2019.

[ABH+22] Melissa Azouaoui, Olivier Bronchain, Clément Hoffmann, Yulia Kuzovkova, Tobias Schneider, and François-Xavier Standaert. Systematic study of decryption and re-encryption leakage: The case of kyber. In *COSADE*, volume 13211 of *Lecture Notes in Computer Science*, pages 236–256. Springer, 2022.

[AHKS22] Amin Abdulrahman, Vincent Hwang, Matthias J. Kannwischer, and Daan Sprenkels. Faster kyber and dilithium on the cortex-m4. In *ACNS*, volume 13269 of *Lecture Notes in Computer Science*, pages 853–871. Springer, 2022.

[BBC+20] Davide Bellizia, Olivier Bronchain, Gaëtan Cassiers, Vincent Grosso, Chun Guo, Charles Momin, Olivier Pereira, Thomas Peters, and François-Xavier Standaert. Mode-level vs. implementation-level physical security in symmetric cryptography - A practical guide through the leakage-resistance jungle. In Daniele Micciancio and Thomas Ristenpart, editors, *Advances in Cryptology - CRYPTO 2020 - 40th Annual International Cryptology Conference, CRYPTO 2020, Santa Barbara, CA, USA, August 17-21, 2020, Proceedings, Part I*, volume 12170 of *Lecture Notes in Computer Science*, pages 369–400. Springer, 2020.

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

[BG14] Shi Bai and Steven D. Galbraith. An improved compression technique for signatures based on learning with errors. In *CT-RSA*, volume 8366 of *Lecture Notes in Computer Science*, pages 28–47. Springer, 2014.

[BGS15] Sonia Belaïd, Vincent Grosso, and François-Xavier Standaert. Masking and leakage-resilient primitives: One, the other(s) or both? *Cryptogr. Commun.*, 7(1):163–184, 2015.

[BMPS21] Olivier Bronchain, Charles Momin, Thomas Peters, and François-Xavier Standaert. Improved leakage-resistant authenticated encryption based on hardware AES coprocessors. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2021(3):641–676, 2021.

[CGMZ21] Jean-Sébastien Coron, François Gérard, Simon Montoya, and Rina Zeitoun. High-order polynomial comparison and masking lattice-based encryption. *IACR Cryptol. ePrint Arch.*, page 1615, 2021.

[CJRR99] Suresh Chari, Charanjit S. Jutla, Josyula R. Rao, and Pankaj Rohatgi. Towards sound approaches to counteract power-analysis attacks. In *CRYPTO*, volume 1666 of *Lecture Notes in Computer Science*, pages 398–412. Springer, 1999.

[CS20] Gaëtan Cassiers and François-Xavier Standaert. Trivially and efficiently composing masked gadgets with probe isolating non-interference. *IEEE Trans. Inf. Forensics Secur.*, 15:2542–2555, 2020.

[CS21] Jean-Sébastien Coron and Lorenzo Spignoli. Secure shuffling in the probing model. *IACR Cryptol. ePrint Arch.*, page 258, 2021.

[DFH+16] Stefan Dziembowski, Sebastian Faust, Gottfried Herold, Anthony Journault, Daniel Masny, and François-Xavier Standaert. Towards sound fresh re-keying with hard (physical) learning problems. In *CRYPTO (2)*, volume 9815 of *Lecture Notes in Computer Science*, pages 272–301. Springer, 2016.

[DLL+17] Léo Ducas, Tancrède Lepoint, Vadim Lyubashevsky, Peter Schwabe, Gregor Seiler, and Damien Stehlé. CRYSTALS - dilithium: Digital signatures from module lattices. *IACR Cryptol. ePrint Arch.*, page 633, 2017.

[DMMS21] Sébastien Duval, Pierrick Méaux, Charles Momin, and François-Xavier Standaert. Exploring crypto-physical dark matter and learning with physical rounding towards secure and efficient fresh re-keying. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2021(1):373–401, 2021.

[FO99] Eiichiro Fujisaki and Tatsuaki Okamoto. Secure integration of asymmetric and symmetric encryption schemes. In *CRYPTO*, volume 1666 of *Lecture Notes in Computer Science*, pages 537–554. Springer, 1999.

[HHP+21] Mike Hamburg, Julius Hermelink, Robert Primas, Simona Samardjiska, Thomas Schamberger, Silvan Streit, Emanuele Strieder, and Christine van Vredendaal. Chosen ciphertext k-trace attacks on masked CCA2 secure kyber. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2021(4):88–113, 2021.

[HOM06] Christoph Herbst, Elisabeth Oswald, and Stefan Mangard. An AES smart card implementation resistant to power analysis attacks. In *ACNS*, volume 3989 of *Lecture Notes in Computer Science*, pages 239–252, 2006.

[HS01] Nick Howgrave-Graham and Nigel P. Smart. Lattice attacks on digital signature schemes. *Des. Codes Cryptogr.*, 23(3):283–290, 2001.

[HSST22] Julius Hermelink, Silvan Streit, Emanuele Strieder, and Katharina Thieme. Adapting belief propagation to counter shuffling of ntts. *IACR Cryptol. ePrint Arch.*, page 555, 2022.

[ISW03] Yuval Ishai, Amit Sahai, and David A. Wagner. Private circuits: Securing hardware against probing attacks. In *CRYPTO*, volume 2729 of *Lecture Notes in Computer Science*, pages 463–481. Springer, 2003.

[KPP20] Matthias J. Kannwischer, Peter Pessl, and Robert Primas. Single-trace attacks on keccak. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2020(3):243–268, 2020.

[KRSS] Matthias J. Kannwischer, Joost Rijneveld, Peter Schwabe, and Ko Stoffelen. PQM4: Post-quantum crypto library for the ARM Cortex-M4. https://github.com/mupq/pqm4.

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

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

[MGTF19] Vincent Migliore, Benoît Gérard, Mehdi Tibouchi, and Pierre-Alain Fouque. Masking dilithium - efficient implementation and side-channel evaluation. In *ACNS*, volume 11464 of *Lecture Notes in Computer Science*, pages 344–362. Springer, 2019.

[MUTS22] Soundes Marzougui, Vincent Ulitzsch, Mehdi Tibouchi, and Jean-Pierre Seifert. Profiling side-channel attacks on dilithium: A small bit-fiddling leak breaks it all. *IACR Cryptol. ePrint Arch.*, page 106, 2022.

[Nat] National Institute of Standards and Technology. Post-quantum cryptography standardization. https://csrc.nist.gov/Projects/Post-Quantum-Cryptography/Post-Quantum-Cryptography-Standardization.

[PPM17] Robert Primas, Peter Pessl, and Stefan Mangard. Single-trace side-channel attacks on masked lattice-based encryption. In *CHES*, volume 10529 of *Lecture Notes in Computer Science*, pages 513–533. Springer, 2017.

[PSV15] Olivier Pereira, François-Xavier Standaert, and Srinivas Vivek. Leakage-resilient authentication and encryption from symmetric cryptographic primitives. In Indrajit Ray, Ninghui Li, and Christopher Kruegel, editors, *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, October 12-16, 2015*, pages 96–108. ACM, 2015.

[REB+22] Prasanna Ravi, Martianus Frederic Ezerman, Shivam Bhasin, Anupam Chattopadhyay, and Sujoy Sinha Roy. Will you cross the threshold for me? generic side-channel assisted chosen-ciphertext attacks on ntru-based kems. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2022(1):722–761, 2022.

[RJH+18] Prasanna Ravi, Mahabir Prasad Jhanwar, James Howe, Anupam Chattopadhyay, and Shivam Bhasin. Side-channel assisted existential forgery attack on dilithium - A NIST PQC candidate. *IACR Cryptol. ePrint Arch.*, page 821, 2018.

[RPBC20] Prasanna Ravi, Romain Poussier, Shivam Bhasin, and Anupam Chattopadhyay. On configurable SCA countermeasures against single trace attacks for the NTT - A performance evaluation study over kyber and dilithium on the ARM cortex-m4. In *SPACE*, volume 12586 of *Lecture Notes in Computer Science*, pages 123–146. Springer, 2020.

[RRCB20] Prasanna Ravi, Sujoy Sinha Roy, Anupam Chattopadhyay, and Shivam Bhasin. Generic side-channel attacks on cca-secure lattice-based PKE and kems. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2020(3):307–335, 2020.

[SETA22] Chao Sun, Thomas Espitau, Mehdi Tibouchi, and Masayuki Abe. Guessing bits: Improved lattice attacks on (EC)DSA with nonce leakage. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2022(1):391–413, 2022.

[UBS21] Balazs Udvarhelyi, Olivier Bronchain, and François-Xavier Standaert. Security analysis of deterministic re-keying with masking and shuffling: Application to ISAP. In *COSADE*, volume 12910 of *Lecture Notes in Computer Science*, pages 168–183. Springer, 2021.

[USS+20] Florian Unterstein, Marc Schink, Thomas Schamberger, Lars Tebelmann, Manuel Ilg, and Johann Heyszl. Retrofitting leakage resilient authenticated encryption to microcontrollers. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2020(4):365–388, 2020.

[UXT+22] Rei Ueno, Keita Xagawa, Yutaro Tanaka, Akira Ito, Junko Takahashi, and Naofumi Homma. Curse of re-encryption: A generic power/em analysis on post-quantum kems. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2022(1):296–322, 2022.

[VMKS12] Nicolas Veyrat-Charvillon, Marcel Medwed, Stéphanie Kerckhof, and François-Xavier Standaert. Shuffling against side-channel attacks: A comprehensive study with cautionary note. In *ASIACRYPT*, volume 7658 of *Lecture Notes in Computer Science*, pages 740–757. Springer, 2012.

A Dilithium Description

Key generation. The key generation is defined in [DLL+17, Fig 4.] and is recalled in Algorithm 6. Initially, a random bit string ζ is created and used to generate three seeds ρ, ς and K thanks to the hash function $\mathtt{H}$. A public matrix $\mathbf{A}$ for which all coefficients are uniform in $\mathbb{Z}_q$ is generated from ρ. Two secret vectors $\mathbf{s}_1 \in S_\eta^l$ and $\mathbf{s}_2 \in S_\eta^k$ are derived from ς. Then, the vector $\mathbf{t} = \mathbf{A}\mathbf{s}_1 + \mathbf{s}_2$ is calculated. This is an instance of MLWE, where $\mathbf{s}_1$ and $\mathbf{s}_2$ are hard to calculate given $\mathbf{A}$ and $\mathbf{t}$. Next, the bit representation of $\mathbf{t}$ is split up into high order bits $\mathbf{t}_1$ and low order bits $\mathbf{t}_0$. Only $\mathbf{t}_1$ will be part of the public key, to keep its size as small as possible. For the same reason the matrix seed ρ is part of the output, rather than the whole matrix $\mathbf{A}$. Lastly, $\rho \| \mathbf{t}_1$ gets hashed to tr. The output is the public key $pk = (\rho, \mathbf{t}_1)$ and the secret key $sk = (\rho, K, tr, \mathbf{s}_1, \mathbf{s}_2, \mathbf{t}_0)$.

Algorithm 6 KeyGen.

1: $\zeta \leftarrow \{0,1\}^{256}$
2: $(\rho, \varsigma, K) = \mathtt{H}(\zeta)$ ▷ $(\rho, \varsigma, K) \in \{0,1\}^{256} \times \{0,1\}^{512} \times \{0,1\}^{256}$
3: $\mathbf{A} = \mathtt{ExpandA}(\rho)$ ▷ $\mathbf{A} \in R^{k \times l}$
4: $(\mathbf{s_1}, \mathbf{s_2}) = \mathtt{ExpandS}(\varsigma)$ ▷ $(\mathbf{s_1}, \mathbf{s_2}) \in S_\eta^l \times S_\eta^k$
5: $\mathbf{t} = \mathbf{A}\mathbf{s_1} + \mathbf{s_2}$
6: $(\mathbf{t_1}, \mathbf{t_0}) = \mathtt{Power2Round}(\mathbf{t}, d)$
7: $tr = \mathtt{H}(\rho \| \mathbf{t_1})$ ▷ $tr \in \{0,1\}^{256}$
8: **return** $pk = (\rho, \mathbf{t_1}), sk = (\rho, K, tr, \mathbf{s_1}, \mathbf{s_2}, \mathbf{t_0})$

Signature. Similarly, we now describe the signing procedure in Algorithm 7. We refer to [DLL+17, Fig 4.] for a more detailed description. The input is the secret key sk and a message M. The message is preprocessed with $\mathtt{H}$ into a bit string μ of fixed length. For deterministic signing, μ is used together with K to produce a seed ρ'. For the randomized version the seed ρ' is generated randomly. This seed and a rejection counter κ (initially set to $\kappa = 0$) are used to sample the secret polynomial $\mathbf{y} \in \tilde{S}_{\gamma_1}^l$ with $\mathtt{ExpandMask}$. Then, the product $\mathbf{w} = \mathbf{A}\mathbf{y}$ is decomposed via division with remainder into $\mathbf{w}_1$ and $\mathbf{w}_0$. The challenge $\tilde{c}$ is the hash of $\mu \| \mathbf{w}_1$. For further calculations, $\tilde{c}$ is converted into a polynomial c that contains strictly τ coefficients set to ± 1 and the others set to zero. This polynomial is then used to calculate $\mathbf{z}$ and $\tilde{\mathbf{r}}$. To ensure the security and correctness of the scheme, two checks are performed:

$$\|\mathbf{z}\|_\infty < \gamma_1 - \beta, \qquad \|\tilde{\mathbf{r}}\|_\infty < \gamma_2 - \beta,$$

where $\beta = \eta \cdot \tau$. If any of the two conditions does not hold, κ is increased and the process starts over (beginning with the sampling of a new $\mathbf{y}$). After successful checks, a hint $\mathbf{h}$ is calculated. This is needed in the verification step to make up for the "lost" information of $\mathbf{t}_0$. Two more checks are performed on $c\mathbf{t}_0$ and $\mathbf{h}$. Again, if these conditions are not met, the signature is rejected and κ is increased. Otherwise, if all checks are successful, the signature $\sigma = (\tilde{c}, \mathbf{z}, \mathbf{h})$ can be output.

Algorithm 7 Sign(sk, M).

$\mathbf{A} = \texttt{ExpandA}(\rho)$
$\mu = \texttt{H}(tr \| M)$ ▷ $\mu \in \{0,1\}^{512}$
$\kappa = 0, (\mathbf{z}, \mathbf{h}) = \perp$
$\rho' = \texttt{H}(K \| \mu)$ (or $\rho' \xleftarrow{\$} \{0,1\}^{512}$ for randomized signing) ▷ $\rho' \in \{0,1\}^{512}$
while $(\mathbf{z}, \mathbf{h}) = \perp$ **do**
  $\mathbf{y} = \texttt{ExpandMask}(\rho', \kappa)$ ▷ $\mathbf{y} \in \tilde{S}^l_{\gamma_1}$
  $\mathbf{w} = \mathbf{Ay}$
  $(\mathbf{w_0}, \mathbf{w_1}) = \texttt{Decompose}(\mathbf{w}, 2\gamma_2)$
  $\tilde{c} = \texttt{H}(\mu \| \mathbf{w_1})$ ▷ $\tilde{c} \in \{0,1\}^{256}$
  $c = \texttt{SampleInBall}(\tilde{c})$ ▷ $c \in B_\tau$
  $\mathbf{z} = \mathbf{y} + c\mathbf{s_1}$
  $\tilde{\mathbf{r}} = \mathbf{w_0} - c\mathbf{s_2}$
  if $\|\mathbf{z}\|_\infty \geq \gamma_1 - \beta$ or $\|\tilde{\mathbf{r}}\|_\infty \geq \gamma_2 - \beta$ **then** $(\mathbf{z}, \mathbf{h}) = \perp$
  else
    $\mathbf{h} = \texttt{MakeHint}(\tilde{\mathbf{r}}, c, \mathbf{t_0}, \mathbf{w_1}, \gamma_2)$
    if $\|c\mathbf{t_0}\|_\infty \geq \gamma_2$ or the # of 1's in $\mathbf{h}$ is greater than ω **then** $(\mathbf{z}, \mathbf{h}) = \perp$
  $\kappa = \kappa + l$
return $\sigma = (\tilde{c}, \mathbf{z}, \mathbf{h})$

B Additional performance numbers

Table 4: Performance of the deterministic Dilithium Level-3 components with software shuffled Keccak: number of clock cycles when running on a `STM32L4R5` and using the TRNG for masking randomness (32-bit randomness every 53 Cycles). Reported numbers are in `kCycles`. The numbers are for a single execution of the component (does not consider repetitions due to rejections).

d	2		4		6		8	
	Masked	Leveled	Masked	Leveled	Masked	Leveled	Masked	Leveled
sign	24,986.8	10,548.1	70,708.4	11,892.0	131,252.0	13,431.8	201,737.5	15,261.0
$\texttt{NTT}(s)$	185.4	185.4	370.8	370.7	556.2	556.3	741.6	741.6
`ExpandA`	2,160.8	2,160.8	2,160.8	2,160.8	2,160.8	2,160.8	2,160.8	2,160.8
$\texttt{H}(\mu \| K)$	367.0	367.2	1,094.5	1,094.9	2,006.2	2,006.8	3,237.9	3,238.6
`ExpandMask`	12,860.2	3,793.3	37,913.5	3,794.4	69,920.0	3,793.0	110,770.7	3,792.5
$\mathbf{Ay}$	382.2	2,459.8	764.4	2,459.8	1,146.6	2,459.8	1,528.7	2,459.7
`Decompose`	2,971.8	134.7	9,443.7	134.8	18,475.8	134.8	27,724.9	134.8
$\mathbf{y} + \mathbf{s}_1 c$	174.9	238.3	349.8	387.4	524.5	536.5	699.5	685.5
$\mathbf{w}_0 - \mathbf{s}_2 c$	209.9	398.3	419.6	584.5	629.4	770.7	839.3	956.7
reject	4,826.7	216.3	16,522.9	216.2	33,062.4	216.2	50,462.1	216.2
umsk	372.8	119.2	1,174.7	195.3	2,258.1	285.4	3,041.5	344.6