IACR Transactions on Cryptographic Hardware and Embedded Systems
ISSN 2569-2925, Vol. 2023, No. 4, pp. 58–79. DOI:10.46586/tches.v2023.i4.58-79

Protecting 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, Eindhoven, Netherlands {firstname.lastname}@nxp.com
[2] UCLouvain, Ottignies-Louvain-la-Neuve, Belgium, {firstname.lastname}@uclouvain.be
[3] Graz University of Technology, Graz, Austria, {firstname.lastname}@iaik.tugraz.at
[4] Lamarr Security Research, Graz, 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 to their physical security requirements. Second, we provide improved gadgets dedicated to Dilithium, taking advantage of recent advances in masking conversion algorithms. Third, we combine these contributions and report performance for side-channel protected Dilithium implementations. 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: CRYSTALS-Dilithium · Lattice-Based Cryptography · Post-Quantum Cryptography · Signatures · Side-Channel Countermeasures · Masking

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 anticipated 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. These include (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.

Licensed under Creative Commons License CC-BY 4.0.

Received: 2023-04-15 Accepted: 2023-06-15 Published: 2023-08-31

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) in order 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 [ABH+22].

To the best of our knowledge, digital signatures, including Dilithium, have received 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 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. The amount of published works appears to be even scarcer when it comes to protecting Dilithium against leakage. To the best of our knowledge, the main contribution comes from Migliore et al. [MGTF19] and presents 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 intermediate computation as sensitive even though they can be explicitly used to recover the secret key. Conversely, others 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. To the best of our knowledge, our work presents the first masked Dilithium design compliant with the third round submission document for all parameter sets.

Second, and following the security requirements of 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.

Finally, we provide a complete benchmark for an ARM Cortex-M4 microcontroller, which includes the evaluation of individual components, their comparison with the ones of Migliore et al., and performance results of full signature generation for deterministic and randomized versions of Dilithium. They highlight the advantages of randomized Dilithium compared to its deterministic variant in the context of physical attacks.

Cautionary note. In an earlier presentation of our results, at the NIST's 4th PQC Standardization Conference, a finer-grain sensitivity analysis distinguishing security against Simple Power Analysis and Differential Power Analysis was proposed [ABC+22]. This analysis was conjectured to enable strongly leveled implementations of Dilithium, where different parts of the implementation use different countermeasures (e.g., shuffling against SPA [VMKS12], masking against DPA [CJRR99, ISW03]). We clarify in Section 3.4 that the possibility to leverage a "hard physical learning problem" similar to [DMMS21] that would back up this conjecture does not hold. As a result, Dilithium has less potential for leveling and its sensitivity analysis can be simplified to a coarser-grain mix of sensitive operations that require DPA protections and non-sensitive ones that can leak in full. We discuss directions to partially restore the interest of a finer-grain leveling in conclusions.

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:

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

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 algorithm 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]. 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 key generation and signature generation algorithms in the following paragraphs. We do not consider the verification, which does not involve long-term secret variables (hence, does not leak sensitive information). Table 1 provides the Dilithium parameters for different NIST security levels.

Key generation. The key generation is defined in [DLL+17, Fig 4.] and is recalled in Algorithm 1. Initially, a random bit string ζ is created and used to generate three seeds ρ, ς and K thanks to the hash function `H`. A public matrix $\mathbf{A}$ for which all coefficients are

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 ± 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

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 1 KeyGen.

```
$\zeta \leftarrow \{0,1\}^{256}$
$(\rho, \varsigma, K) = \mathtt{H}(\zeta)$        ▷ $(\rho, \varsigma, K) \in \{0,1\}^{256} \times \{0,1\}^{512} \times \{0,1\}^{256}$
$\mathbf{A} = \mathtt{ExpandA}(\rho)$        ▷ $\mathbf{A} \in R^{k \times l}$
$(\mathbf{s_1}, \mathbf{s_2}) = \mathtt{ExpandS}(\varsigma)$        ▷ $(\mathbf{s_1}, \mathbf{s_2}) \in S_\eta^l \times S_\eta^k$
$\mathbf{t} = \mathbf{A}\mathbf{s_1} + \mathbf{s_2}$
$(\mathbf{t_1}, \mathbf{t_0}) = \mathtt{Power2Round}(\mathbf{t}, d)$
$tr = \mathtt{H}(\rho \| \mathbf{t_1})$        ▷ $tr \in \{0,1\}^{256}$
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 2. 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 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 `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 in order 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 2 Sign(sk, M).

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

3 Sensitivity analysis

In this section, we analyze Dilithium's key generation and signature generation and discuss the sensitivity of all the variables and functions potentially leading to side-channel attacks. This sensitivity analysis indicates which operations/variables need to be protected against leakage. As mentioned in introduction, we use a coarse-grain taxonomy for this purpose, which is next reflected by color-coded diagrams: Figure 1 and Figure 2, where red (resp., blue) denotes sensitive variables/operations that need security against DPA (resp., variables/operations that do not require side-channel attack protection). Doing so we also compare our analysis to the one previously proposed in [MGTF19].

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] M does not need to be protected, but $\mathbf{s}_1$ and $\mathbf{s}_2$, and K, must be protected in order to avoid side-channel attacks leading to a signature forgery. Next, we detail which other variables must be protected in order to avoid the leakage of long-term sensitive secrets. We start with their sensitivity analysis for the key generation followed by the signing procedure.

3.1 Key generation sensitivity

During key generation, the variable ζ has to be protected since it is the seed for all subsequent values (e.g., K). Similarly, ς has to be protected since it serves as a seed to deterministically generate the long term secrets $\mathbf{s}_1$ and $\mathbf{s}_2$. 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. Alternatively, it can decomposed into $\mathbf{t}_0$ and $\mathbf{t}_1$ so that the public key only contains $\mathbf{t}_1$. This reduces the size of the public key by a factor close to two at the cost of an increased secret key size, which must then contain $\mathbf{t}_0$, and a slightly increased signature size.

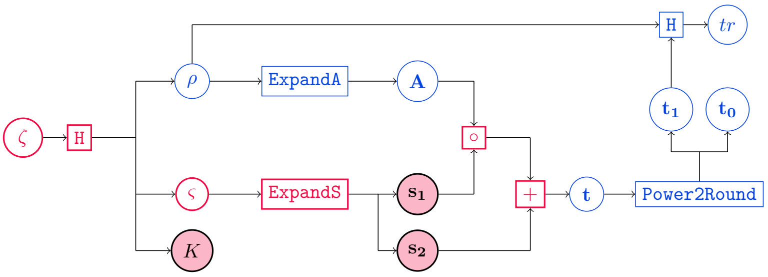


Figure 1: Graphical representation of the key generation. Output: $pk = (\rho, \mathbf{t_1})$, $sk = (\rho, K, tr, \mathbf{s_1}, \mathbf{s_2}, \mathbf{t_0})$. Red: sensitive. Blue: non-sensitive.

3.2 Signature generation sensitivity

All the variables denoted in red in Figure 2 need to remain secret and hence must be secure against DPA. This naturally holds for both secret key components $\mathbf{s_1}$ and $\mathbf{s_2}$, in order to avoid trivial key recovery attacks leading to signature forgeries. Next, 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 or partial knowledge of $\mathbf{y}$ [MUTS22]. A similar analysis applies to $\mathbf{w}_0$ which can lead to the recovery of $\mathbf{s}_2$.[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). This equation is similar to a Learning With Rounding (LWR) instance, where w_1 would be the public rounded value and w_0 would be the error (which cannot leak). For the same reason, ρ' must be protected since it is used as a seed to obtain $\mathbf{y}$. The same holds for K in the deterministic signing case.

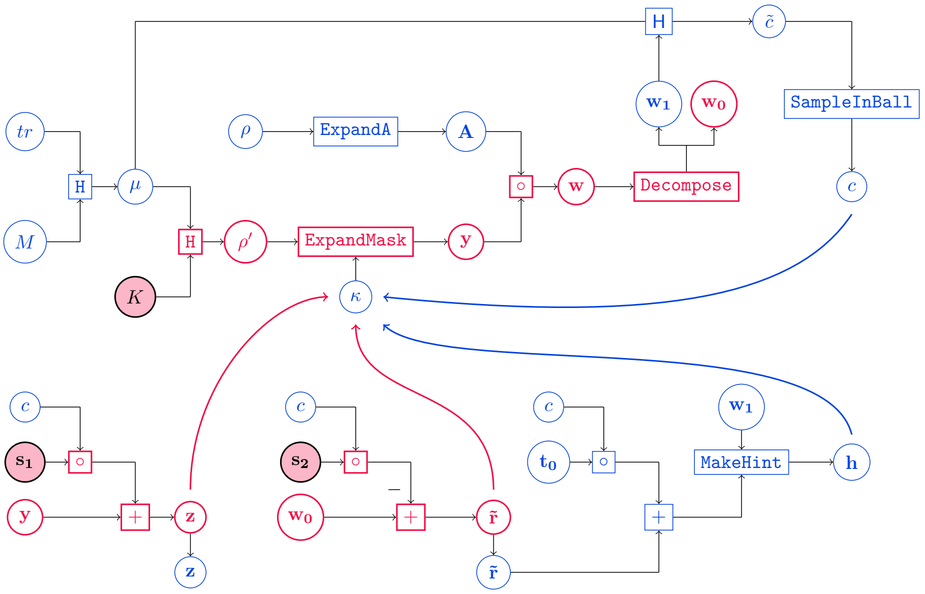


Figure 2: Graphical representation of the signature generation. Input: sk, M, Output: $\sigma = (\tilde{c}, \mathbf{z}, \mathbf{h})$. Curved arrows represent rejection checks. Red: sensitive. Blue: non-sensitive.

Next, when it comes to public or non-sensitive variables, both tr and $\mathbf{t_0}$, the message M, the seed ρ, the hash μ and the matrix $\mathbf{A}$ are public. The sensitivity of the vector of

[2] A recent ePrint report details an attack exploiting the leakage of $\mathbf{w}_0$ [BVC⁺23].

polynomials $\mathbf{w}_1$ in the signing procedure is more delicate to analyze, since it depends on the result of the boundary checks on $\mathbf{z}$ and $\tilde{\mathbf{r}}$. If the boundary checks pass, for example when a signature is accepted, then the zero-knowledge proof of security of Dilithium shows that $\mathbf{w}_1$ does not leak any information. Informally, $\mathbf{w}_1$ can be reconstructed from a valid signature, which in turn can be simulated in zero knowledge, and hence $\mathbf{w}_1$ contains no more information than the signature itself. As a result, Dilithium does not need an explicit LWR hardness assumption in this case, since it is at least as hard as its LWE assumption. When the boundary checks do not pass, the reduction from LWR to LWE does not apply immediately since the distribution on $\mathbf{w}_1$ changes slightly. Leaving $\mathbf{w}_1$ unmasked therefore requires the additional explicit assumption that the corresponding LWR problem is hard. Since the number of rounded bits is significantly higher than the error that is added in the LWE problem, we conjecture that $\mathbf{w}_1$ does not require side-channel countermeasures. The same expectation was also shared by Vadim Lyubashevsky in a personal communication and publicly during RWPQC23. We leave its further investigation, e.g., in the light of the recent [DFPS23], as an interesting scope for further investigations. In order to be conservative, we nevertheless study the option of additionally protecting $\mathbf{w}_1$ in Subsection 6.4. For the rest, the challenge c can be left unprotected since it is derived from a one-way hash of $\mathbf{w}_1$ and public inputs. And once the bound checks on $\mathbf{z}$ and $\tilde{\mathbf{r}}$ (discussed further in the next paragraph) have passed, the hint vector can be made public since it does not contain any sensitive information. Indeed, in the simplified version of Dilithium which does not involve the hints or the public key compression, all information that would be given by the hints is already contained in the returned valid signature and the public key. The checks on $c\mathbf{t_0}$ and $\mathbf{h}$ are needed for correctness only.

Finally, regarding $\mathbf{z}$ and $\tilde{\mathbf{r}}$, both must remain protected until the bound checks on both have passed. This implies the need for a secure bound check algorithm. After successful bound 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:

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

Indeed, for a valid signature, the values $\mathbf{A}$, $\mathbf{z}$, c, $\mathbf{t}$, and $\mathbf{w}_1$ are not 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.

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: $\mathbf{A}\mathbf{y} = \mathbf{w}$. 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 the sensitive $\mathbf{y}$ in less than two minutes on a laptop, with original Dilithium parameters.[4] For level 2, since the

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

[4] 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.

matrix $\mathbf{A}$ is square (of dimensions 4×4) and random, it is most likely invertible.[5] 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, contrary to what was previously proposed 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 $\texttt{Decompose}(\mathbf{w})$. 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 $\texttt{Decompose}(\mathbf{r})$. Also, the $\texttt{MakeHint}$ function works slightly different and takes $\mathbf{r}, c, \mathbf{t}_0$ as input (but produces the same exact output $\mathbf{h}$). In [MGTF19], this $\mathbf{r}$-version is used while the $\tilde{\mathbf{r}}$-version is never mentioned. However, considering the equation:

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

we can see 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 output. So we consider $\tilde{\mathbf{r}}$ as public after the checks on $\mathbf{z}$ and $\tilde{\mathbf{r}}$.

3.4 Differences with [ABC+22]

In this previous version of our results, a finer-grain sensitivity analysis was proposed, suggesting that the computation of $c\mathbf{s}_1 + \mathbf{y} = \mathbf{z}$ could only require security against SPA. It was in particular conjectured that an intermediate attack path targeting this computation would be hard, the investigation of which being left as an open problem. However, it appears that the analogy between the multiplication $c\mathbf{s}_1$ and a variant of "hard physical learning problem" similar to [DMMS21], which would back up this conjecture, does not hold. The problem, already glimpsed in [ABC+22], is that c and $\mathbf{s}_1$ are not uniformly distributed and have small norm, while the hardness of hard physical learning problems leverages uniform secrets so that computing the multiplication leads to modular reductions. Combined with the fact that when the signature is correct, the knowledge on $\mathbf{z}$ can be used to directly transfer information on $\mathbf{y}$ into information on the output of the multiplication, it implies that $\mathbf{y}$ and $\mathbf{z}$ actually need to be protected against DPA. As a result, Dilithium has less opportunities of levelling and its sensitivity analysis can be simplified into the mix of unprotected and DPA protected operations that we now use.

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 in the case of the different parameter sets of Dilithium.

Concretely, these gadgets are essentially relying on standard approaches tailored to the Dilithium use case and most of our optimizations are obtained from carefully selecting the type of masking (i.e., Boolean or arithmetic) We in particular rely on the recently improved masking conversion proposed in [BC22] for this purpose.

[5] More precisely, 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, it almost always holds that the determinant of $\mathbf{A}$ is non-zero and an inverse matrix exists.

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 converting 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 the implementation of these algorithms.

In this work, 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. This means that PINI gadgets can be composed (without refresh) and the resulting circuit is probing secure. Since the new gadgets we propose can be expressed as a 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 4. 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.

Algorithm 3 $\texttt{SecUnMask}_k^d\left(\boldsymbol{x}^{B,k}\right)$

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

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

Algorithm 4 $\texttt{SecLeq}_\psi^d\left(\boldsymbol{x}^{B,k}\right)$

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

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

Correctness. We next detail the correctness of Algorithm 4 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 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 4 is PINI if b is public.*

Proof. `SecUnMask` is PINI as a consequence of [CGMZ23, Lemma 2].[6]. Therefore, if b is public, Algorithm 4 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 5 and Algorithm 6. We note that the $\texttt{SecAdd}^d_{k+1}$ 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 5 describes $\texttt{SecBoundCheck}^d_{q,\lambda_0,\lambda_1}\left(\boldsymbol{x}^{A_q}\right)$ which returns a bit b if the input arithmetic sharing $\boldsymbol{x}^{A_q}$ satisfies the property $-\lambda_0 \leq x \leq \lambda_1 \mod q$.

Algorithm 5 $\texttt{SecBoundCheck}^d_{q,\lambda_0,\lambda_1}\left(\boldsymbol{x}^{A_q}\right)$

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.

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

Correctness The first step in Algorithm 5 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`.

Proposition 2. *Algorithm 5 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 4 is PINI by Proposition 1. Hence, Algorithm 5 is PINI since it is the composition of PINI gadgets. □

[6] `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.

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 6 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 6 $\texttt{SecSampleModp}^d_{q,\phi_0,\phi_1}\left(\boldsymbol{x}_0^{B,k}, \boldsymbol{x}_1^{B,k}, \dots\right)$

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 $-\phi_0 \leq x \leq \phi_1 \mod q$.

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

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 6 is to convert its uniformly distributed input bits into shares $\boldsymbol{x}_i^{B,k}$ while the obtained x is strictly larger than $\phi_0 + \phi_1$. This inequality is checked by leveraging SecLeq described in Algorithm 4. 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 6 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 6 is itself PINI. □

Usage in Dilithium. SecSampleModp is used during both for key generation and signing. 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, $k \in \{18, 20\}$ depends 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.[7] 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 7 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.

Algorithm 7 $\texttt{SecDecompose}^{d}_{q,\alpha}\left(\boldsymbol{w}^{A_q}\right)$

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 $w = \alpha w_1 + w_0 \mod q$.

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

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 its 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:

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

[7] 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`.

for which a masked version at any order is presented in [CGMZ23]. 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 $w \mod q$.[8]

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 7 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:

$$\begin{aligned} b &= w_1 + \alpha^{-1} \cdot (w_0 - \frac{\alpha}{2}) \mod q, &(5)\\ &= w_1 + 16 \cdot (\frac{\alpha}{2} - w_0) \mod q. &(6)\end{aligned}$$

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 $\texttt{SecA2BModp}_q^d$ and just keeping the 4 LSBs of the output.[9]

Proposition 4. *Algorithm 7 is PINI if w_1 is public.*

Proof. If w_1 is public, Algorithm 7 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 describe the implementations for both the deterministic and the randomized versions of Dilithium. 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. In order to prevent side-channel attacks, we follow our previous sensitivity analysis that distinguishes sensitive and non-sensitive operations, and we make use of masking with the gadgets presented in Section 4 and the underlying masked additions and conversion gadgets such as `SecA2BModp`, `SecAdd`, `SecAddModp` or `SecB2AModp` for all sensitive ones. 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 arithmetic to arithmetic masking conversion from q' to q to perform the addition with masked $\mathbf{y}$. We leave the study of such a trade-off for future works.

5.1 Deterministic Dilithium

We use a masked Keccak for $\mathtt{H}(K||\mu)$ and within `ExpandMask`. In `ExpandMask`, the randomness generation in `SecSampleModp` (see Algorithm 6 Line-3) is performed with a call to the

[8] [CGMZ23] only considers δ equals a power of two. We take δ as an arbitrary positive number.

[9] We note that only the LSBs of the $\texttt{SecA2BModp}_q^d$ have to be explicitly computed. As a result, this can save several `SecAnd` when the $\texttt{SecA2BModp}_q^d$ from [BC22] is used.

masked XOF. 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 from Algorithm 7 with the appropriate parameters given in Subsection 4.5. 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 5. Eventually, similarly to SecLeq in Algorithm 5, we unmask the public signature $\mathbf{z}$ and $\tilde{\mathbf{r}}$ using the SecUnMask gadget once all the bound checks are passed, to maintain probing security.

5.2 Randomized Dilithium

The implementation we consider of the randomized version is similar to the deterministic one previously described. The main difference lies in the sampling of the randomness $\mathbf{y}$ within ExpandMask. The randomized version of Dilithium enables more freedom in the generation of the uniform polynomial vector $\mathbf{y}$ compared to the deterministic one. For the deterministic version, the randomness sampling in Algorithm 6 is based on secured XOF (Keccak) which can be a performance bottleneck (as detailed in Table 2). For the randomized version, a first option (which follows the specifications but does not bring performance improvements) is to generate ρ' with a TRNG and to use the masked XOF to derive $\mathbf{y}$. Alternatively, one can directly generate the shares of $\mathbf{y}$ with the TRNG. This does not follow the specifications of Dilithium but saves the cost of a masked XOF and does not weaken the security of Dilithium as $\mathbf{y}$ remains uniform. We next evaluate this option.[10]

6 Benchmarks

In this section, we report the performances of the Dilithium implementations described in Section 5. We first detail the benchmarking setup used for this purpose. Second, we report the performance improvements provided by the new gadgets of Section 4 compared to the ones of [MGTF19]. Then, we evaluate the cost of each individual operation in Dilithium's signature generation (without considering the rejections). Based on this, 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 general conclusions apply to all security levels.

6.1 Benchmarking setup

In order to evaluate the execution time of our implementations, 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 for the generation of the randomness masking as well as for the ExpandMask in the randomized version of Dilithium that we evaluate.

6.2 Gadgets improvements

We first compare the gadgets presented in Section 4 and the ones proposed by Migliore et al. in [MGTF19], and we report the results in Figure 3. 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.

[10] Yet another alternative, in case of weak TRNG, is to generate shares of $\mathbf{y}$ by applying the unmasked XOF to TRNG outputs. This option saves the cost of masking the XOF.

As a result, the implementation of [MGTF19] we consider does not contain extra refresh gadgets (as it is PINI). 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 isolated in our implementations thanks to bitslicing.

`SecSampleModp`. Both versions of `SecSampleModp` as used in the signature generation are similar. The only difference is that the subtraction with ϕ_0 is performed with Boolean masking in [MGTF19] and with arithmetic masking in Algorithm 6. 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 3b.

`SecBoundCheck`. Our `SecBoundCheck` also simplifies the one proposed in [MGTF19] where a `SecA2BModp` is performed followed by two `SecAdd`'s.[11] 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 3d.

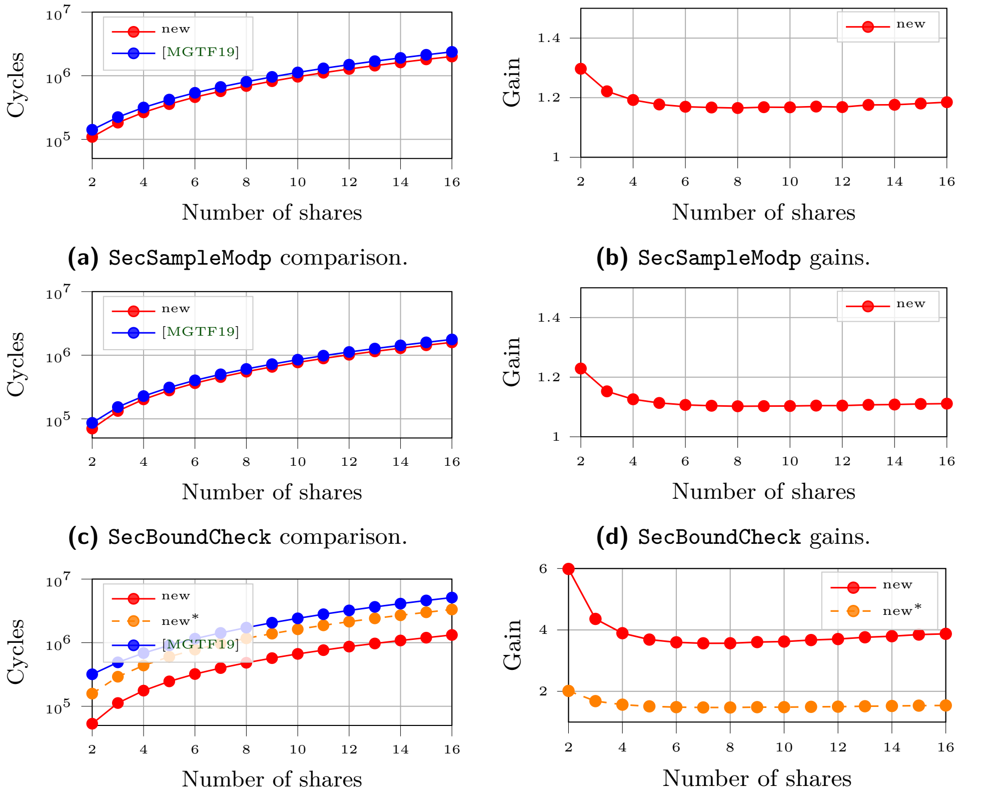
`SecDecompose`. Finally, we compare the two implementations of `SecDecompose`. Interestingly, the main improvement comes from the fact that we first extract $\mathbf{w}_1$ efficiently and then unmask it to compute $\mathbf{w}_0$. This improvement relies on the fact that the higher order bits (e.g., $\mathbf{w}_1$) of the `SecDecompose` gadget is considered as sensitive by Migliore et al. while it is not necessary as detailed in the revisited sensitivity analysis detailed in Section 3.3. 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, and runs ≈ 3.8 times faster.

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 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 vs. randomized performances

The performances of each operation within both versions of Dilithium are reported in Table 2. We observe that the randomized signature generation is more efficient than the deterministic one. For two shares, 24 005 kCycles are needed for the deterministic version vs. only 14 282 kCycles for the randomized one. Hence, randomization offers an improvement by a factor ≈ 1.68. Similarly, for 8 shares, the randomized version is $\approx 1.77\times$ faster than the deterministic one. The run time of unprotected Dilithium3 signature generation is 3224 kCycles [KRSS]. Hence, the two-share version is 4.3× slower than the unprotected implementation in the randomized case and 7.32× in the deterministic case. Most of the difference between the two Dilithium versions is due to `ExpandMask`, which is composed of two parts as detailed in Algorithm 6. The first one is sampling the uniform $\mathbf{y}$ in Boolean masking. This operation is performed with a masked `XOF` in the deterministic case, and with the on-board TRNG in the randomized case. The second part is to perform a `SecB2AModp` in order to produce an arithmetic sharing. This operation is similar for both cases. In the deterministic case, the `ExpandMask` represents 56 % of the total run

[11] Only `SecBoundCheck` is described for Boolean sharing in [MGTF19]. Here we assume that a `SecA2BModp` is performed before the end to match Algorithm 5 specifications.



(a) `SecSampleModp` comparison.

(b) `SecSampleModp` gains.

(c) `SecBoundCheck` comparison.

(d) `SecBoundCheck` gains.

(e) `SecDecompose` comparison.

(f) `SecDecompose` gains.

Figure 3: Comparison between our new gadgets and [MGTF19] for NIST Level-3 parameters. The label `new`* corresponds to the version where $\mathbf{w}_1$ remains masked.

Table 2: Performances of the masked Dilithium Level-3 components for randomized and deterministic versions: number of clock cycles are given when running on a STM32L4R5 and using the TRNG for generating the masking randomness (32-bit randomness every 53 Cycles). Reported numbers are in kCycles. The numbers are for a single execution of the component (and do not consider repetitions due to rejections). Rand. *: The vector polynomial $\mathbf{y}$ is sampled from the TRNG and not from a XOF.

d	2		4		6		8	
	Deter.	Rand.*	Deter.	Rand.*	Deter.	Rand.*	Deter.	Rand.*
Sign	23,613.8	13,891.4	68,557.9	39,064.5	128,280.9	73,905.4	199,567.4	111,515.4
NTT(s)	185.4	185.4	370.7	370.7	556.3	556.2	741.7	741.7
ExpandA	2,160.7	2,160.7	2,160.7	2,160.7	2,160.7	2,160.7	2,160.7	2,160.7
H($K\|\|\mu$)	370.1	0.0	1,126.7	0.0	2,082.8	0.0	3,377.8	0.0
ExpandMask	12,551.1	3,198.6	38,068.8	9,702.2	70,788.2	18,514.6	113,185.0	28,498.6
SecB2AModp	3,350.2	3,104.8	9,993.8	9,516.7	18,967.1	18,235.5	29,081.0	28,128.4
Sampling	9,193.9	79.7	28,062.1	165.9	51,802.5	253.7	84,079.6	339.0
$\mathbf{Ay}$	382.2	382.3	764.4	764.4	1,146.7	1,146.6	1,528.8	1,528.8
Decompose	2,369.0	2,369.1	8,142.3	8,142.3	16,382.8	16,376.8	25,076.9	25,080.8
H($\mu\|\|\mathbf{w}_1$)	94.9	94.9	94.9	94.9	94.9	94.9	94.9	94.9
$\mathbf{y} + \mathbf{s}_1 c$	174.9	174.9	349.7	349.7	524.6	524.6	699.5	699.5
$\mathbf{w}_0 - \mathbf{s}_2 c$	209.9	210.0	419.6	419.7	629.4	629.5	839.3	839.4
SecBoundCheck	4,354.6	4,354.6	15,472.8	15,472.8	31,234.4	31,221.3	48,376.4	48,386.4
UnMask	378.3	378.4	1,185.7	1,185.7	2,260.7	2,260.6	3,046.8	3,047.0

time from which 74 % are due to the masked XOF. As the randomized Dilithium does not require this masked XOF, only 25 % are imputable to ExpandMask.

The cost of the other operations are similar for both versions. Concretely, the overall cost is dominated by the operations that have quadratic overheads in the number of shares (even for $d = 2$). These operations are H($K||\mu$), ExpandMask, SecDecompose, SecBoundCheck and UnMask. We note that for the deterministic version, the most expensive operation is ExpandMask, while it is SecBoundCheck for the randomized version. Additionally, the cost of polynomial arithmetic (NTT(s), $\mathbf{y} + \mathbf{s}_1 c$ and $\mathbf{w}_0 - \mathbf{s}_2 c$) is limited. As expected, the cost of the public matrix expansion ExpandA remains constant with the number of shares. Eventually, we note that the SecB2AModp is slightly more expensive in the deterministic case, as it includes the linear overheads needed in order to map the output of the masked XOF into the correct bitslice representation. This operation is not needed in the randomized case as the output of the TRNG already has the appropriate layout.

More generally, we also stress that the randomized version does not allow the adversary to average traces for the same inputs, which is beneficial for security. The combination of these observations and performance gains naturally calls for considering the randomized Dilithium in application contexts where side-channel attacks are a concern.

6.4 Protecting $\mathbf{w}_1$ with masking

We finally discuss the cost of a more conservative implementation keeping $\mathbf{w}_1$ masked before the bound checks on $\mathbf{z}$ and $\tilde{\mathbf{r}}$ are passed. The resulting block diagram for signature generation is given in Figure 4, where the only difference with Figure 2 is that both $\mathbf{w}_1$ and H($\mu||\mathbf{w}_1$) are in red (i.e., sensitive). This change implies that $\mathbf{w}_1$ must remain masked in SecDecompose (see Algorithm 7). Hence, SecDecompose is adapted to keep $\mathbf{w}_1$ masked by performing a SecB2AModp on its Boolean sharing (instead of unmasking). The computation of $\mathbf{w}_0$ can then be applied share-wise similarly as in Algorithm 7, Line-9. The resulting increase in the run time of SecDecompose is detailed in Figure 3.

The impact of protecting $\mathbf{w}_1$ on the full signature is detailed in Table 3. The main difference with Table 2 is that the cost of SecDecompose increases as discussed above. A second difference is that the hash function H($\mu||\mathbf{w}_1$) now also needs to be masked. As a result, this hashing represents a total of 9 % of the run time for deterministic signatures,

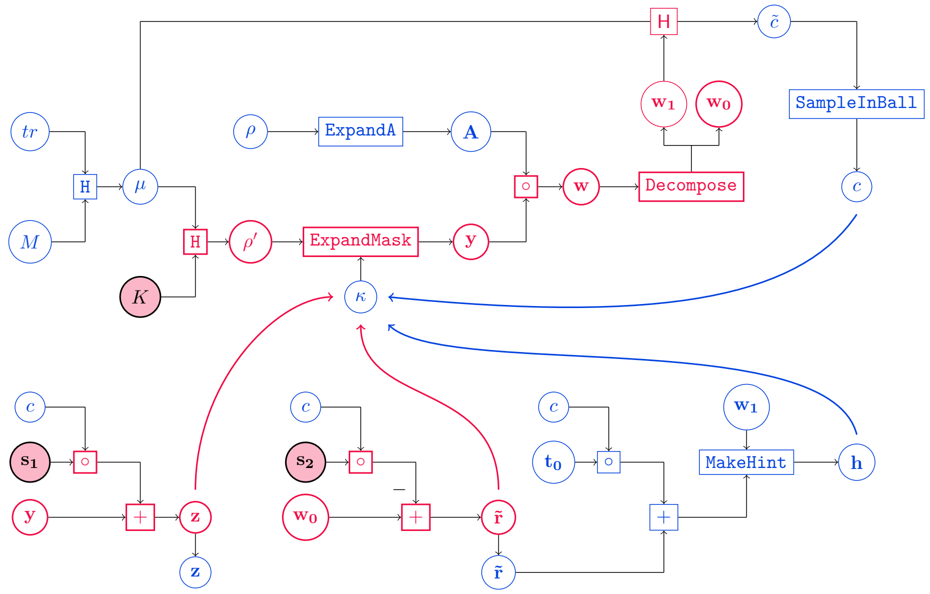


Figure 4: Graphical representation of the signing procedure with masked $\mathbf{w}_1$, taking as input sk, M and outputting $\sigma = (\tilde{c}, \mathbf{z}, \mathbf{h})$. Curved arrows represent rejection checks. Red filled circles: sensitive variables. Blue: no side-channel protection required.

and 13.7 % for randomized ones. Hence, protecting $\mathbf{w}_1$ decreases the benefit of randomized signatures, from 66 % of the run time when $\mathbf{w}_1$ does not need to be protected to 56 % when it has. Overall, the run time of deterministic (resp., randomized) signatures increases by a factor 1.3 (resp., 1.53) when we need to protect $\mathbf{w}_1$ with masking.

Table 3: Performance of the masked Dilithium Level-3 components for randomized and deterministic versions with masked $\mathbf{w}_1$: number of clock cycles when running on a STM32L4R5 and using the TRNG for generating the 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). Rand. *: The vector polynomial $\mathbf{y}$ is sampled from the TRNG and not from a XOF.

d	2		4		6		8	
	Deter.	Rand.*	Deter.	Rand.*	Deter.	Rand.*	Deter.	Rand.*
Sign	30,254.8	20,532.1	88,912.9	59,512.8	166,748.8	112,397.1	259,743.1	171,692.1
NTT(s)	185.4	185.4	370.7	370.8	556.2	556.2	741.6	741.7
ExpandA	2,160.7	2,160.7	2,160.7	2,160.7	2,160.7	2,160.7	2,160.7	2,160.7
H($K\|\|\mu$)	370.1	0.0	1,126.7	0.0	2,082.8	0.0	3,377.8	0.0
ExpandMask	12,551.3	3,198.6	38,057.4	9,712.6	70,783.2	18,514.4	113,185.7	28,495.5
SecB2AModp	3,350.3	3,104.8	9,982.3	9,528.9	18,962.0	18,235.4	29,081.7	28,126.1
Sampling	9,193.9	79.9	28,062.1	164.1	51,802.5	253.7	84,079.6	338.4
$\mathbf{Ay}$	382.3	382.2	764.4	764.4	1,146.6	1,146.6	1,528.7	1,528.8
Decompose	6,527.5	6,527.4	20,767.0	20,813.1	40,448.2	40,447.9	61,779.3	61,790.4
H($\mu\|\|\mathbf{w}_1$)	2,576.9	2,576.9	7,860.8	7,860.8	14,514.4	14,514.5	23,560.8	23,560.8
$\mathbf{y} + \mathbf{s}_1 c$	174.9	174.9	349.7	349.7	524.5	524.6	699.5	699.4
$\mathbf{w}_0 - \mathbf{s}_2 c$	209.9	209.9	419.7	419.8	629.5	629.5	839.3	839.3
SecBoundCheck	4,354.7	4,354.7	15,447.0	15,471.9	31,221.1	31,221.1	48,380.3	48,389.1
UnMask	584.8	584.7	1,832.4	1,832.6	3,493.6	3,493.6	4,708.8	4,708.9

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. Second, our new masking 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 and we propose the first masking gadgets that are compatible with all the Dilithium parameter sets. Overall, our analysis and benchmark highlight that the randomized variant of Dilithium evaluated in this paper provides notably better performances, thanks to additional flexibility in the sampling of random values. In addition, it also offers a smaller side-channel attack surface as signatures cannot be repeated. 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 lead to a number of natural open problems. First, they highlight that for now, the leveling concept (i.e., the idea of protecting different parts of an implementations with different countermeasures, in order to limit the overheads) cannot be fully exploited for Dilithium, as initially thought (see the cautionary note in Section 1). For example, most of the signature operations in our implementations need to be secure against DPA, which requires (expensive) masked gadgets. Hence it is a natural open question to find out whether more leveled implementations could be obtained, which could be considered in different fashions. A light leveling option, directly applicable to Dilithium, would be to try exploiting that even when masking, all operations may not leak in a similar manner. For example, one could try leveraging the recent observation that prime masking is more resilient to low-noise leakages than Boolean masking, and study whether the number of shares in the Boolean and arithmetic masking used for protecting Dilithium against leakage could be leveled [MMMS23]. A more ambitious direction would be to study how to enable a stronger leveling again (i.e., mixing operations that require security against SPA and operations that require security against DPA). One potential direction could be to rely on hard physical learning problems like introduced in [DMMS21], possibly at the cost of some design tweaks in Dilithium (e.g., in order to deal with the challenges raised by the manipulation of non-uniform and low-norm secrets). In general, designing a PQ signature scheme with a better performance vs. side-channel security tradeoff appears as an interesting long-term goal. Besides, our work only focuses on side-channel attacks and therefore raises the question of how to additionally protect Dilithium against fault attacks, and whether its randomized version also provides (security or performance) benefits in this context, which is yet another interesting direction for further research.

The source codes of the implementations described in this work are available from the following link: https://github.com/uclcrypto/pqm4_masked/tree/master.

Acknowledgments.

François-Xavier Standaert is a senior research associate of the Belgian Fund for Scientific Research (FNRS-F.R.S.). This work and its presentation have been funded in parts by the ERC Consolidator Grant number 724725 (acronym SWORD) and the ERC Advanced Grant number 101096871 (acronym BRIDGE).

References

[ABC$^+$22] Melissa Azouaoui, Olivier Bronchain, Gaëtan Cassiers, Clément Hoffmann, Yulia Kuzovkova, Joost Renes, Markus Schönauer, Tobias Schneider, François-Xavier Standaert, and Christine van Vredendaal. Leveling dilithium against leakage: Revisited sensitivity analysis and improved implementations. *Fourth PQC Standardization Conference*, 2022.

[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.

[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.

[BVC$^+$23] Alexandre Berzati, Andersson Calle Viera, Maya Chartouni, Steven Madec, Damien Vergnaud, and David Vigilant. A practical template attack on crystals-dilithium. *IACR Cryptol. ePrint Arch.*, page 50, 2023.

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

[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 wire shuffling in the probing model. In *CRYPTO (3)*, volume 12827 of *Lecture Notes in Computer Science*, pages 215–244. Springer, 2021.

[DFPS23] Julien Devevey, Pouria Fallahpour, Alain Passelègue, and Damien Stehlé. A detailed analysis of fiat-shamir with aborts. *IACR Cryptol. ePrint Arch.*, page 245, 2023.

[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.

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

[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.

[MMMS23] Loïc Masure, Pierrick Méaux, Thorben Moos, and François-Xavier Standaert. Effective and efficient masking with low noise using small-mersenne-prime ciphers. In *EUROCRYPT (4)*, volume 14007 of *Lecture Notes in Computer Science*, pages 596–627. Springer, 2023.

[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.

[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.

[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.