Leveling Dilithium against Leakage
Revisited Sensitivity Analysis and Improved Implementations

Melissa Azouaoui¹, Olivier Bronchain¹,², Gaëtan Cassiers²,³,⁴, Clément Hoffmann², Yulia Kuzovkova¹, Joost Renes¹, Tobias Schneider¹, Markus Schönaier¹, François-Xavier Standaert² and Christine van Vredendaal¹

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 \[\text{FO99}\] 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 \[\text{RRCB20, REB}^+22, \text{UXT}^+22\]. 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 \[\text{RPBC20, HHP}^+21, \text{KPP20, PPM17}\].

This variety of threats implies a large attack surface leading to significant overheads when protecting PQC KEM’s against SCA’s \[\text{ABH}^+22\].

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. \[\text{RJH}^+18\] that shows that to achieve existential forgery an attacker only requires knowledge of one part of the secret key in Dilithium, namely \(s_1\). Marzougui et al. \[\text{MUTS22}\] exploit leakage of the zero coefficients in the secret signing nonce \(y\) for multiple signatures and recover the secret key by leveraging least squares regression and integer linear programming. Liu et al. \[\text{LZS}^+21\] 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 \(y\) 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 \[\text{HS01}\] and very recently improved by Sun et al. \[\text{SETA22}\]. 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. \[\text{MGTF19}\]. 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. \[\text{MGTF19}\]. 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. \[\text{ABH}^+22\], 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 \( z \mod \alpha \) to mean the unique integer \( z' \) in \( \left[ -\frac{\alpha}{2}, \frac{\alpha}{2} \right] \) (resp., \( \left[ -\frac{\alpha - 1}{2}, \frac{\alpha - 1}{2} \right] \)) with \( z \equiv z' \mod \alpha \) if \( \alpha \) is even (resp., odd). The notation \( z \mod^{\pm} \alpha \) implies that all the coefficients in \( z \) are given with \( \mod^{\pm} \alpha \).

With this, we can define the following norms on \( \mathbb{Z}_q \), \( R \) and \( R^k \) respectively:

\[
\|z\|_\infty = |z \mod^{\pm} q|, \quad \|p\|_\infty = \max_i \|p_i\|_\infty, \quad \|w\|_\infty = \max_i \|w_i\|_\infty,
\]

with \( z \in \mathbb{Z}_q \), \( p \in R \), \( p_i \) being the \( i \)-th coefficient of \( p \), \( w \in R^k \) and \( w_i \) being the \( i \)-th polynomial in \( w \). Additionally, we define \( S_\eta = \{ w \in R : \|w\|_\infty \leq \eta \} \) and \( \tilde{S}_\eta = \{ w \mod^{\pm} 2\eta : w \in R \} \). This means that the coefficients of an element in \( S_\eta \) or \( \tilde{S}_\eta \) are in the range \( [-\eta, \eta] \) or \( [-\eta, \eta] \), respectively. We use the notation \( x \leftarrow X \) whenever we assign a uniformly random element of a set \( X \) 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.
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 \( A \) can also be leaked since it is deterministically derived from \( \rho \). A similar status holds for some parts of the secret key \( sk := (\rho, K, tr, s_1, s_2, t_0) \), 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 \( t_0 \) can be leaked as well. Indeed, the Dilithium security proofs consider \( t \) (hence \( t_0 \) and \( t_1 \)) to be public [DLL +17]. Furthermore, we do not consider message recovery attacks on the message \( M \). As a result, only the vector of polynomials \( s_1 \) and \( s_2 \), 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 \( \zeta \) has to be protected as it is the seed for all subsequent values (e.g., \( K \)). Similarly, \( \varsigma \) has to be protected as well, since it serves as a seed to deterministically generate the long term secrets \( s_1 \) and \( s_2 \). 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.

As a result \( t \) could be fully part of the public key. In order to reduce the size of the public key, \( t \) is decomposed into \( t_0 \) and \( t_1 \). The public key only contains \( t_1 \) instead of \( t \). 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 \( t_0 \).
Figure 1: Graphical representation of the key generation. Output: $pk = (\rho, t_1)$, $sk = (\rho, K, tr, s_1, s_2, t_0)$. 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 $t_0$, the message $M$, the seed $\rho$, the hash $\mu$ and the public matrix $A$ are public. However, the vector of polynomials $y$ is sensitive and must be protected. Indeed, given a valid signature $\sigma = (\tilde{c}, z, h)$, the secret vector $s_1$ can be recovered from $z = y + cs_1$ for known $y$ [MUTS22]. A similar analysis applies to $w_0$ which can lead to the recovery of $s_2$. As a result, the vector of polynomials $w$ must be protected: $w_0$ is directly derived from $w$, and it is possible to solve the system of equations $Ay = w$ for known $A$ and $w$ to recover $y$ in most cases (see Section 3.3 for details). For the same reason, $\rho'$ must be protected since it is used as a seed to obtain $y$.

Continuing with the signing procedure, the vector of polynomials $w_1$ 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 $w_1$ and public inputs. Then, before the bound check is performed, both $z$ and $\tilde{r}$ 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 $z$ this is trivial, as it is part of the signature. In the case of $\tilde{r}$, this can be shown by the equation:

$$Az - ct = w - cs_2 = \alpha w_1 + \tilde{r}.$$ 

Indeed, for a valid signature, none of the values $A$, $z$, $c$, $t$, and $w_1$ are sensitive and $\alpha$ is a known parameter of the algorithm. Therefore, $\tilde{r}$ can be computed using only public values, so there is no need to keep it protected after a successful signing process. A public $\tilde{r}$ is quite handy, because it allows us to compute the hint $h$ 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 $\text{ExpandMask}$ is indistinguishable from random since it is derived deterministically from the secret random seed $\rho'$. Therefore, the subsequent sensitive values (namely $y, w, w_0$) are ephemeral, since they are obtained from the seed $\rho'$ and fixed public matrix $A$ without any other data allowing a DPA. It then remains that a SPA can be performed on $\text{ExpandMask}$ based on the limited number of varying values for $\kappa$, which depends on whether the deterministic or randomized version of Dilithium is considered. (The same holds for $Ay$ based on the fact that every polynomial in $y$ is multiplied by

\(^2\)The value $w_1$ does not leak information about $y$ after the signature is successfully calculated. This is due to the LWR (Learning With Rounding) assumption. The assumption is independent of rejection, and so $w_1$ can be unmasked even when the corresponding signature gets rejected. See also [BG14].

\(^3\)This refers to the rejection checks on $z$ and $\tilde{r}$. The one on $h$ is not sensitive.
Figure 2: Graphical representation of the signing procedure, taking as input $sk, M$ and outputting $\sigma = (\tilde{c}, z, h)$. 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.

In the deterministic case, the bound on the number of $\kappa$ values that can be observed depends on the number of rejections and can reach $\approx 20$ on average.\(^4\) In the randomized case, it is tightly bound to 1. We emphasize that despite $y$ 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 $c \cdot s_1$ should be protected against DPA while $y$ and $z$ should be protected against SPA (a similar situation holds with $s_2, w_0$ and $\tilde{r}$). This leads to the additional observation that leakage on the SPA-protected $y$ and $z$ could be used on order to gain information about the DPA-protected multiplication $c \cdot s_1$. We next provide heuristic arguments why this attack path should not lead to simple attacks against Dilithium.

First, turning information on the (21-bit) $z$ into information on the (8-bit) $c \cdot s_1$ is affected by the algorithmic noise of the (20-bit) $y$. For illustration, we show on Figure 3 the mutual information of the polynomial multiplication output given the Hamming weight of $y$ and $z$, with the bitsize of $y$ used as a security parameter. As this bitsize increases, the mutual information decreases significantly below the mutual information of $\approx 2.5$ that would be observed for an ephemeral leakage on $c \cdot s_1$ without algorithmic noise.

Second, and as will be discussed next, $y$ and $z$ 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 $c s_1$ by a factor 256 [VMKS12].

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

\(^4\)For Level-3 parameters, $y$ is sampled on average 5.1 times with $\text{ExpandMask}$. More precisely, within each $\text{ExpandMask}$ execution, $\rho'$ is hashed together with $l = 5$ independent inputs to generate all the polynomials in $y$. This results on average in a total of 25.5 hashes of $\rho'$ with independent inputs.
8 Leveling Dilithium against Leakage

Figure 3: Mutual information leakage on $c \cdot s_1$ given the Hamming weight of $y$ and $z$. 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 $w$ and $\tilde{r}$ slightly differ, which we discuss in the following.

Protecting $w$. First we look at $w$, in particular at the system of equations that produces it: $Ay = w$. It is possible to solve this system for $y$, if the matrix $A$ has one more row than columns. This is the case for NIST security levels 3 and 5, where $A$ has dimensions $6 \times 5$ and $8 \times 7$ respectively. Even a simple solver is able to compute $y$ in less than two minutes on a laptop, with original Dilithium parameters. For level 2, since the matrix $A$ is square (of dimensions $4 \times 4$) and random, it is most likely invertible. Hence, with knowledge of $w$, $y$ can be computed simply as $y = A^{-1} \cdot w$. This shows that $w$ must be protected (or rather $w_0$, since $w_1$ is public), contrary to the approach in [MGTF19].

Unmasking $\tilde{r}$. Before we look at $\tilde{r}$ 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 $w_1$ from Decompose($w$). Then, instead of $\tilde{r} = w_0 - cs_2$ it computes $r = w - cs_2$. The rejection check on $\tilde{r}$ is done on $r_0$, which comes from Decompose($r$). Also, the MakeHint function works slightly different and takes $r, c, t_0$ as input (but produces the same exact output $h$). In [MGTF19] however, the $r$-version is used while the $\tilde{r}$-version is never mentioned. This is not a problem when it comes to comparing our security assessments. To see this, consider the equation:

$$r = w - cs_2 = \alpha w_1 + w_0 - cs_2 = \alpha w_1 + \tilde{r}.$$  

First, the polynomials of the first column of $A$ are reduced to monic form via Gaussian elimination on the coefficients. Then, the fact that $a_{ij} X^k \cdot X^{n-k} = -a_{ij} X^0 (\mod X^n + 1)$ 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.

5Since the entries of $A$ are elements of $R$, and $R$ contains $q^n = 8380417$ elements, it is highly unlikely that two rows of $A$ are linearly dependent. Therefore, the determinant of $A$ is almost always non-zero and an inverse matrix exists.
This shows that $r$ and $\tilde{r}$ can be calculated from each other using the public values $w_1$ and $\alpha$. So any consideration regarding the sensitivity classification of one of these values, automatically applies to the other one as well. In [MGTF19], the value $r$ is never unmasked which means that the calculation of the hint $h$ must be protected against side-channel attacks. But as we explained above, $\tilde{r}$ can be recreated from public values after a valid signature is output. So we consider $\tilde{r}$ as public after the checks on $z$ and $\tilde{r}$.

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 $x^{B,k}$ and satisfies the property that $x = \bigoplus_{i=0}^{d-1} x^{B,k}_i$ where $x^{B,k}_i$ is the $i$-th share of $x$. The notation $x^{B,k}[j]$ 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 $x^{A,q}$ such that $x = \sum_{i=0}^{d-1} x^{A,q}_i \mod q$ where $x^{A,q}_i$ 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 $\text{SecA2BModp}^{d,q}$. Similarly to convert from Boolean masking to arithmetic masking, we use $\text{SecB2AModp}^{d,q}$. Eventually, we also leverage the gadget $\text{SecAddModp}^{d,q}$ 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 $\text{SecLeq}^{d,q}(x^{B,k})$ 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 the bound $\psi$.

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

**Input:** Boolean sharing \(x^{B,k}\) with \(0 \leq x < 2^k\).

**Output:** Output the \(k\)-bit unmasked value \(x\).

1: \(y^{B,k} \leftarrow \text{Refresh}_d^d(x^{B,k})\) \hspace{1cm} \(\triangleright\) Refresh based on the ISW multiplication [CS21, Algorithm 3].
2: \(x \leftarrow \bigoplus_{i=1}^{d-1} x_i^{B,k}\)

Algorithm 2 SecLeq\(_d^d\) (\(x^{B,k}\))

**Input:** Boolean sharing \(x^{B,k}\) 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: \(x^{r^{B,k+1}} \leftarrow \text{SecAdd}_{k+1}^d(x^{B,k}2^{k+1} - \psi - 1)\)
2: \(b \leftarrow \text{SecUnMask}_d^d(x^{r^{B,k+1}[k]}))\)

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 \(\psi\).

**Proposition 1.** Algorithm 2 is PINI if \(b\) is public.

**Proof.** SecUnMask is PINI as a consequence of [CGMZ21, Lemma 2]. Therefore, if \(b\) is public, Algorithm 2 is a composition of PINI gadgets. \(\square\)

**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 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 SecBoundCheck\(_{q,\lambda_0,\lambda_1}^d\) (\(x^{A_x}\)) which returns a bit \(b\) if the input arithmetic sharing \(x^{A_x}\) satisfies the property \(-\lambda_0 \leq x < \lambda_1 \mod q\).

**Correctness.** The first step in Algorithm 3 is to add \(\lambda_0\) 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 \(x'^{A_x}\) of \(x'\) is converted to a Boolean sharing \(x'^{B,k}\) 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 \(\lambda_0\) and \(\lambda_1\) are positive integers, we ensure that \(\lambda_0 + \lambda_1 \geq 0\). The returned bit by SecBoundCheck is the one returned by SecLeq. 

\(^7\text{refresh} \) 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 SecBoundCheck\(d_{q,\lambda_0,\lambda_1}(x^{A_4})\)

Input: Arithmetic sharing \(x^{A_4}\), 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. \(x^{A_4} \leftarrow x^{A_4} + \lambda_0 \mod q\)
2. \(x^{B,k} \leftarrow \text{Sec2BMod}_q^d(x^{A_4})\)
3. \(b \leftarrow \text{SecLeq}_{\lambda_0+\lambda_1}(x^{B,k})\)

Proposition 2. Algorithm 3 is PINI.

Proof. The first addition is applied only on the first share hence PINI. SecA2BMod\(d\) 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 \(\|z\|_\infty < \gamma_1 - \beta\) and \(\|\overline{z}\|_\infty < \gamma_2 - \beta\), where \(\gamma_1, \gamma_2\) and \(\beta\) 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 \((x_0^B, x_1^B, \ldots, \).

Algorithm 4 SecSampleModp\(_d^d_{q,\phi_0,\phi_1}(x_0^B, x_1^B, \ldots,\)

Input: Bounds \(\phi_0\) and \(\phi_1\) with \(\phi_0 \geq 0, \phi_1 \geq 0\) and \(\phi_0 + \phi_1 < q\).
Output: Arithmetic sharing \(x^{A_4}\) 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 \(\neg \text{SecLeq}_{\phi_0,\phi_1}(x_i^B,k)\) do
   4. \(i \leftarrow i + 1\)
5. \(x^{A_4} \leftarrow \text{SecE2AMod}_q^d(x_i^B,k)\)
6. \(x^{A_4}[0] \leftarrow x^{A_4}[0] - \phi_0 \mod q\)

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 \(x_i^B,k\) 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 \(x^{A_4}\). 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 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 \( s_1 \) or \( s_2 \) 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 SecSampleModp\( (\eta, \eta, \eta) \). For these parameters, rejections can occur and a fresh \( x \) passes the SecLeq check with probability \( \frac{1}{2} \) and \( \frac{9}{16} \), respectively. Second, during ExpandMask signature generation, a coefficient \( x \) of \( y \) is sampled such that \( -\gamma_1 < x \leq \gamma_1 \) where \( \gamma_1 \) 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 SecSampleModp\( (\gamma_1, \gamma_1) \). For these parameters, no resampling of \( x \) is required. Indeed, SecLeq\( \gamma_1, 1 \) 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 \( w_0 = w \mod \alpha \). 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 \( w^{A_k} \). Then, \( w_0^{A_k} \) is obtained by computing \( w_0^{A_k} = w^{A_k} - \alpha \cdot w_1 \mod q \). To the best of our knowledge, there is no generic and efficient method for a masked division to compute \( w_0^{A_k} \) divided by \( \alpha \) 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:

\[
\left\lfloor \frac{\alpha w_1 + w_0}{\alpha} \right\rfloor = \lfloor (\alpha w_1 + w_0) \cdot \frac{-\alpha^{-1}}{q} \rfloor, \\
\approx \lfloor (\alpha w_1 + w_0) \frac{-\alpha^{-1}}{q} \rfloor,
\]

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

\[
\text{Compress}(x, \delta, q) = \left\lfloor \frac{x \cdot \delta}{q} \right\rfloor \mod \delta,
\]

\(^8\)An alternative solution is to leverage arithmetic to arithmetic conversion. It would require to first generate a random sharing \( x^{A_{q_0 + q_1}} \) resulting in a uniform \( x \) modulus \( q_0 + q_1 \). Then, that sharing of \( x \) must be converted to arithmetic sharing with modulus \( q \) such as \( x^{A_k} \). Yet to our knowledge, there exists no arithmetic to arithmetic conversion more efficient than the combination of SecLeq and SecB2AModp.
Algorithm 5 SecDecompose$^d_{q,\alpha}$ ($w^A_q$)

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

1: if NIST Level 3 or Level 5 then
2: $b^A_q \leftarrow w^A_q + \gamma_2 \mod q$
3: $b^A_q \leftarrow \alpha^{-1} \cdot b^A_q - 1 \mod q$
4: $b^{B,k} \leftarrow \text{SecA2BModp}_q^d(b^A_q)$
5: $w_1^{B,k'} \leftarrow b^{B,k'}[[0,k']]$
6: else \hspace{1cm} \begin{comment} NIST Level 2 \end{comment}
7: $w_1^{B,k'} \leftarrow \text{SecCompress}^d_{\gamma,-\alpha-1}(w^A_q)$
8: $w_1 \leftarrow \text{SecUnMask}_{\gamma}^d(w_1^{B,k'})$
9: $w_0^A_q \leftarrow w^A_q - \alpha \cdot w_1 \mod q$

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 $w \mod q$.

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:

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

which can be alternatively expressed as:

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

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

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 SecA2BModp$_q^d$ 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. \qed

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 $\delta$ equal to a power of two. We take $\delta$ as an arbitrary positive number.

$^{10}$We note that only the LSBs of the SecA2BModp$_q^d$ have to be explicitly computed. As a result, this can save several SecAnd when the SecA2BModp$_q^d$ 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 \texttt{SecA2BModp}, \texttt{SecAdd}, \texttt{SecAddModp} or \texttt{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 \cite{BC22}. Eventually, we use the same masked Keccak as the one provided in \cite{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 \( s_1 \circ c \) proposed in \cite{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 \( y \). 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 \cite{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. \cite{RPBC20} present different techniques leading to a trade-off between cost and security. Attacks against this shuffling countermeasure have been studied in \cite{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 \cite{RPBC20}, named the \textit{Coarse-Full-Shuffled} strategy. We directly re-use the code provided along with \cite{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 \cite{UBS21}.\textsuperscript{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 foreseeable feature). As discussed in \cite{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 \( H(K||\mu) \) and within \texttt{ExpandMask}. In \texttt{ExpandMask}, the randomness

\textsuperscript{11}The smallest permutation considered for polynomial operations is 256, which is significantly larger than the permutations over 25 independent operations studied in \cite{UBS21}.
generation in \texttt{SecSampleModp} (see Algorithm 4 Line-3) is performed with a call to the secure XOF (based on Keccak). The multiplication \(Ay\) is performed on each of the shares of \(y\) independently by leveraging the optimized arithmetic operations in [KRSS]. For the \(w\) decomposition, we leverage the new gadget \texttt{SecDecompose} presented in Algorithm 5 with the appropriate parameters given in Subsection 4.5. Similarly, the protected rejections \(\|z\|_\infty < \gamma_1 - \beta\) and \(\|\tilde{r}\|_\infty < \gamma_2 - \beta\) are implemented thanks to \texttt{SecBoundCheck} presented in Algorithm 3. Eventually, similarly to \texttt{SecLeq} in Algorithm 3, we unmask the public signature \(z\) and \(\tilde{r}\) using the \texttt{SecUnMask} gadget, in order to maintain probing security.

5.4 Randomized Dilithium

The randomized version of Dilithium enables a larger degree of freedom in the generation of the uniform vector of polynomial \(y\) 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 \(y\) 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 \(y\) 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 \texttt{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 \(y\). A subsequent \texttt{SecB2AModp} must be applied to obtain an arithmetic sharing of \(y\), 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 \(y\) 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

\[12\text{For completeness, we also report in Supplementary Material B the performances of Dilithium signature generation when a custom software shuffled Keccak is used within ExpandMask.}\]
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 $\varphi_0$ 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 $\frac{1}{2}$, 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. 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 $\approx 1.1$, 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 $w_1$ efficiently and then unmask it to compute $w_0$. This improvement relies on the sensitivity detailed in Section 3.3: $w_1$ 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 ($\approx 10$) additions and finally performs a SecB2AModp to obtain the arithmetic sharing of $w_0$. The new gadget only requires a single SecA2BModp and some share-wise operations with arithmetic masking. Overall, the new gadget runs $\approx 3.8$ times faster.

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

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.
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 $\approx 1.2$.

### 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, $24986\text{kCycles}$ are needed for the fully masked implementation vs. only $6757\text{kCycles}$ for the leveled implementation. Hence, leveling offers an improvement by a factor $\approx 3.7$. 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).

<table>
<thead>
<tr>
<th>Component</th>
<th>Masked</th>
<th>Leveled</th>
<th>Masked</th>
<th>Leveled</th>
<th>Masked</th>
<th>Leveled</th>
<th>Masked</th>
<th>Leveled</th>
</tr>
</thead>
<tbody>
<tr>
<td>sign</td>
<td>24,986.8</td>
<td>6,757.8</td>
<td>70,708.4</td>
<td>8,106.7</td>
<td>131,252.0</td>
<td>9,641.8</td>
<td>201,737.5</td>
<td>11,471.5</td>
</tr>
<tr>
<td>NTT(s)</td>
<td>185.4</td>
<td>185.4</td>
<td>370.8</td>
<td>370.7</td>
<td>556.2</td>
<td>556.3</td>
<td>741.6</td>
<td>741.6</td>
</tr>
<tr>
<td>ExpandAdd</td>
<td>2,160.8</td>
<td>2,160.8</td>
<td>2,160.8</td>
<td>2,160.8</td>
<td>2,160.8</td>
<td>2,160.8</td>
<td>2,160.8</td>
<td>2,160.8</td>
</tr>
<tr>
<td>H(K</td>
<td></td>
<td>µ)</td>
<td>367.0</td>
<td>367.2</td>
<td>1,094.5</td>
<td>1,094.9</td>
<td>2,006.2</td>
<td>2,006.8</td>
</tr>
<tr>
<td>ExpandMask</td>
<td>12,860.2</td>
<td>2,459.8</td>
<td>764.4</td>
<td>2,459.8</td>
<td>1,146.6</td>
<td>2,459.8</td>
<td>1,528.7</td>
<td>2,459.7</td>
</tr>
<tr>
<td>Ay</td>
<td>2,971.8</td>
<td>134.7</td>
<td>9,443.7</td>
<td>134.8</td>
<td>18,475.8</td>
<td>134.8</td>
<td>27,724.9</td>
<td>134.8</td>
</tr>
<tr>
<td>Decompose</td>
<td>174.9</td>
<td>238.3</td>
<td>349.8</td>
<td>387.4</td>
<td>524.5</td>
<td>536.5</td>
<td>699.5</td>
<td>685.5</td>
</tr>
<tr>
<td>y + s₁c</td>
<td>209.9</td>
<td>398.3</td>
<td>419.6</td>
<td>584.5</td>
<td>629.4</td>
<td>770.7</td>
<td>839.3</td>
<td>956.7</td>
</tr>
<tr>
<td>w₀ − s₂c</td>
<td>4,826.7</td>
<td>216.3</td>
<td>16,522.9</td>
<td>216.2</td>
<td>33,062.4</td>
<td>216.2</td>
<td>50,462.1</td>
<td>216.2</td>
</tr>
<tr>
<td>reject</td>
<td>372.8</td>
<td>119.2</td>
<td>1,174.7</td>
<td>195.3</td>
<td>2,258.1</td>
<td>285.4</td>
<td>3,041.5</td>
<td>344.6</td>
</tr>
<tr>
<td>umsk</td>
<td>382.2</td>
<td>2,459.8</td>
<td>764.4</td>
<td>2,459.8</td>
<td>1,146.6</td>
<td>2,459.8</td>
<td>1,528.7</td>
<td>2,459.7</td>
</tr>
</tbody>
</table>

- **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 **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 **y + s₁c** and **w₀ − s₂c**. In the fully masked case, the addition of **y** and **w₀** is performed independently on all the shares, whereas the leveled implementation only requires a single shuffled addition on the first share of **s₁c**.

### 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) **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 $\text{kCycles}$. The numbers are for a single execution of the component (does not consider repetitions due to rejections).

<table>
<thead>
<tr>
<th>$d$</th>
<th>Masked 2</th>
<th>Leveled 2</th>
<th>Masked 4</th>
<th>Leveled 4</th>
<th>Masked 6</th>
<th>Leveled 6</th>
<th>Masked 8</th>
<th>Leveled 8</th>
</tr>
</thead>
<tbody>
<tr>
<td>sign</td>
<td>15,375.7</td>
<td>6,459.2</td>
<td>42,050.6</td>
<td>7,074.4</td>
<td>78,880.8</td>
<td>7,703.7</td>
<td>117,273.8</td>
<td>8,301.9</td>
</tr>
<tr>
<td>NTT(s)</td>
<td>185.4</td>
<td>185.4</td>
<td>370.7</td>
<td>370.7</td>
<td>556.2</td>
<td>556.2</td>
<td>741.7</td>
<td>741.6</td>
</tr>
<tr>
<td>ExpandA</td>
<td>2,160.8</td>
<td>2,160.7</td>
<td>2,160.8</td>
<td>2,160.7</td>
<td>2,160.7</td>
<td>2,160.7</td>
<td>2,160.7</td>
<td>2,160.7</td>
</tr>
<tr>
<td>$\text{H}(K</td>
<td>\mu)$</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>ExpandMask</td>
<td>3,594.3</td>
<td>72.1</td>
<td>10,342.5</td>
<td>72.1</td>
<td>19,555.1</td>
<td>72.1</td>
<td>29,547.5</td>
<td>72.1</td>
</tr>
<tr>
<td>$\text{Ay}$</td>
<td>382.3</td>
<td>2,459.5</td>
<td>764.4</td>
<td>2,459.5</td>
<td>1,146.6</td>
<td>2,459.5</td>
<td>1,528.8</td>
<td>2,459.6</td>
</tr>
<tr>
<td>Decompose</td>
<td>2,977.9</td>
<td>134.6</td>
<td>9,445.6</td>
<td>134.7</td>
<td>18,475.4</td>
<td>134.7</td>
<td>27,724.6</td>
<td>134.7</td>
</tr>
<tr>
<td>$y + s_1c$</td>
<td>174.9</td>
<td>238.2</td>
<td>349.7</td>
<td>387.2</td>
<td>524.6</td>
<td>536.3</td>
<td>699.5</td>
<td>685.4</td>
</tr>
<tr>
<td>$w_0 - s_2c$</td>
<td>210.0</td>
<td>398.2</td>
<td>419.7</td>
<td>584.2</td>
<td>629.5</td>
<td>770.3</td>
<td>839.4</td>
<td>956.5</td>
</tr>
<tr>
<td>reject</td>
<td>4,841.9</td>
<td>216.1</td>
<td>16,528.4</td>
<td>216.1</td>
<td>33,062.2</td>
<td>216.1</td>
<td>50,457.0</td>
<td>216.1</td>
</tr>
<tr>
<td>umsk</td>
<td>372.7</td>
<td>119.2</td>
<td>1,174.5</td>
<td>195.2</td>
<td>2,257.7</td>
<td>285.3</td>
<td>3,041.3</td>
<td>344.5</td>
</tr>
</tbody>
</table>

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 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


Algorithm 6. Initially, a random bit string $\beta$ where $\eta$ version the seed $24$ starts over (beginning with the sampling of a new $y$). After successful checks, a hint $h$ is calculated. This is needed in the verification step to make up for the “lost” information of $t_0$. Two more checks are performed on $ct_0$ and $h$. Again, if these conditions are not met, the signature is rejected and $\kappa$ is increased. Otherwise, if all checks are successful, the signature $\sigma = (\tilde{c}, z, h)$ can be output.
Algorithm 7 Sign($sk, M$).

1: $A = \text{ExpandA}()$ \\
2: $\mu = H(tr\|M)$ \hspace{1cm} $\triangleright \mu \in \{0, 1\}^{512}$ \\
3: $\kappa = 0, (z, h) = \perp$ \hspace{1cm} $\triangleright$ $\nu \leftarrow \{0, 1\}^{512}$ for randomized signing \\
4: while $(z, h) = \perp$ do \\
5: $\rho' = H(K||\mu)$ (or $\rho \leftarrow \{0, 1\}^{512}$) \hspace{1cm} $\triangleright \rho' \in \{0, 1\}^{512}$ \\
6: $y = \text{ExpandMask}(\rho', \kappa)$ \hspace{1cm} $\triangleright y \in S_{\gamma_1}$ \\
7: $w = Ay$ \\
8: $(w_0, w_1) = \text{Decompose}(w, 2\gamma_2)$ \\
9: $\hat{c} = H(\mu||w_1)$ \hspace{1cm} $\triangleright \hat{c} \in \{0, 1\}^{256}$ \\
10: $c = \text{SampleInBall}(\hat{c})$ \hspace{1cm} $\triangleright c \in B_r$ \\
11: $z = y + cs_1$ \\
12: $\hat{r} = w_0 - cs_2$ \\
13: if $\|z\|_{\infty} \geq \gamma_1 - \beta$ or $\|w\|_{\infty} \geq \gamma_2 - \beta$ then $(z, h) = \perp$ \\
14: else \\
15: $h = \text{MakeHint}(\hat{r}, c, t_0, w_1, \gamma_2)$ \\
16: if $\|ct_0\|_{\infty} \geq \gamma_2$ or the # of 1’s in $h$ is greater than $\omega$ then $(z, h) = \perp$ \\
17: $\kappa = \kappa + l$ \\
18: return $\sigma = (\hat{c}, z, h)$

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).

<table>
<thead>
<tr>
<th>$d$</th>
<th>sign</th>
<th>Masked</th>
<th>Leveled</th>
<th>Masked</th>
<th>Leveled</th>
<th>Masked</th>
<th>Leveled</th>
<th>Masked</th>
<th>Leveled</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>24.986.8</td>
<td>10,548.1</td>
<td>70.708.4</td>
<td>11.892.0</td>
<td>131.252.0</td>
<td>13,431.8</td>
<td>201.737.5</td>
<td>15,261.0</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>185.4</td>
<td>185.4</td>
<td>370.8</td>
<td>370.7</td>
<td>556.2</td>
<td>556.3</td>
<td>741.6</td>
<td>741.6</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>2,160.8</td>
<td>2,160.8</td>
<td>2,160.8</td>
<td>2,160.8</td>
<td>2,160.8</td>
<td>2,160.8</td>
<td>2,160.8</td>
<td>2,160.8</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>367.0</td>
<td>367.2</td>
<td>1,094.5</td>
<td>1,094.9</td>
<td>2,006.2</td>
<td>2,006.8</td>
<td>3,237.9</td>
<td>3,238.6</td>
<td></td>
</tr>
<tr>
<td>H($\mu</td>
<td></td>
<td>K$)</td>
<td>12,860.2</td>
<td>3,793.3</td>
<td>3,793.3</td>
<td>3,794.4</td>
<td>69,920.0</td>
<td>3,793.0</td>
<td>110,770.7</td>
</tr>
<tr>
<td>ExpandMask</td>
<td>382.2</td>
<td>2,459.8</td>
<td>764.4</td>
<td>2,459.8</td>
<td>1,146.6</td>
<td>2,459.8</td>
<td>1,528.7</td>
<td>2,459.7</td>
<td></td>
</tr>
<tr>
<td>Ay</td>
<td>2,971.8</td>
<td>134.7</td>
<td>9,443.7</td>
<td>134.8</td>
<td>18,475.8</td>
<td>134.8</td>
<td>27,724.9</td>
<td>134.8</td>
<td></td>
</tr>
<tr>
<td>y + s_1c</td>
<td>209.9</td>
<td>209.9</td>
<td>419.6</td>
<td>584.5</td>
<td>629.4</td>
<td>770.7</td>
<td>890.3</td>
<td>956.7</td>
<td></td>
</tr>
<tr>
<td>reject</td>
<td>4,826.7</td>
<td>216.3</td>
<td>16,522.9</td>
<td>216.2</td>
<td>33,062.4</td>
<td>216.2</td>
<td>50,462.1</td>
<td>216.2</td>
<td></td>
</tr>
<tr>
<td>umsk</td>
<td>372.8</td>
<td>119.2</td>
<td>1,174.7</td>
<td>195.3</td>
<td>2,258.1</td>
<td>285.4</td>
<td>3,041.5</td>
<td>344.6</td>
<td></td>
</tr>
</tbody>
</table>