Fault Attacks Sensitivity of Public Parameters in the Dilithium Verification

Andersson Calle Viera[1,2], Alexandre Berzati[1], and Karine Heydemann[1,2]

[1] Thales DIS, France
andersson.calle-viera, alexandre.berzati,
karine.heydemann@thalesgroup.com
[2] Sorbonne Université, CNRS, Inria, LIP6, F-75005 Paris, France

Keywords: Dilithium · Digital signature · Fault Attacks · Side-channel attacks · Post-quantum cryptography · Lattice-based cryptography

Abstract. This paper presents a comprehensive analysis of the verification algorithm of the CRYSTALS-Dilithium, focusing on a C reference implementation. Limited research has been conducted on its susceptibility to fault attacks, despite its critical role in ensuring the scheme's security. To fill this gap, we investigate three distinct fault models - randomizing faults, zeroizing faults, and skipping faults - to identify vulnerabilities within the verification process. Based on our analysis, we propose a methodology for forging CRYSTALS-Dilithium signatures without knowledge of the secret key. Instead, we leverage specific types of faults during the verification phase and some properties about public parameters to make these signatures accepted. Additionally, we compared different attack scenarios after identifying sensitive operations within the verification algorithm. The most effective requires potentially fewer fault injections than targeting the verification check itself. Finally, we introduce a set of countermeasures designed to thwart all the identified scenarios rendering the verification algorithm intrinsically resistant to the presented attacks.

1 Introduction

Shor's algorithm [34], capable of breaking current cryptosystems [32,12], has underscored the urgency for post-quantum cryptography (PQC). As the third round of the National Institute of Standards and Technology (NIST) has concluded [1], four new post-quantum public key schemes are set to be standardized by 2024. Although estimates on the availability of sufficiently large quantum computers remain uncertain, there is a concerted effort by academia and industry to be ready when these algorithms are standardized. It is noteworthy that of the three digital signature schemes chosen, two are based on hard problems over structured lattices. This work focuses on CRYSTALS-Dilithium [3], hereafter referred to as Dilithium, which is the NIST recommended standard for quantum-safe digital signatures [1].

The effective and secure implementation of cryptographic algorithms on current hardware platforms poses a challenge, as it might be vulnerable to fault attacks (FA) and side-channel attacks (SCA). Securing embedded cryptographic applications against such attacks is essential but complex. It requires not only to consider a large set of attacks but also the potential impact on performances of the deployed protections. Although Dilithium has been designed to be resistant to timing attacks, recent work showed that its implementations are likely to leak secret informations [15,25,30,23,2,21,29]. Fault attacks significantly threaten the security of cryptographic systems, potentially undermining the integrity and confidentiality of sensitive data. In this paper, we focus on the analysis of Dilithium's verification algorithm - a crucial component of signature schemes - with a particular emphasis on its sensitivity to fault attacks. Unlike well-established signatures, like RSA or DSA and their variants, the verification algorithm of Dilithium has yet to be precisely analyzed [33,27,7]. By exploring theoretical attack vectors on the verification algorithm, we aim to provide a comprehensive understanding of the vulnerabilities associated with this procedure.

Our Contributions. In this work, we present three properties allowing the generation of forged Dilithium signatures that, when combined with appropriate faults, can bypass the verification without requiring the knowledge of the secret key. To facilitate a deeper understanding of the verification algorithm's sensitivity to fault attacks, we meticulously investigate each identified operation and assess its susceptibility to four common theoretical fault models. From this investigation, we detail three scenarios, the most realistic ones, but we also discuss other potential locations. We comprehensively summarize the analyzed locations and the required corresponding fault models in Tab. 1. In addition, we present a set of relevant dedicated countermeasures aiming at mitigating the different scenarios.

2 Preliminaries

In this section, we present the essential background on Dilithium to better understand the various attack paths presented. We also give a short summary of existing fault attacks and signature forgery methods relative to Dilithium.

Notations: Let us note $\mathbb{Z}_q$ the ring of integers modulo q and $\mathbb{Z}_q[X] = (\mathbb{Z}/q\mathbb{Z})[X]$ the set of polynomials with integer coefficients modulo q. We define $R = \mathbb{Z}[X]/(X^n+1)$ the ring of polynomials with integer coefficients, reduced by the cyclotomic polynomial X^n+1 and $R_q = \mathbb{Z}_q[X]/(X^n+1)$ when considering integer coefficients modulo q. Elements in R_q are denoted by lowercase letters, while elements in R_q^k or R_q^l are denoted by bold lowercase letters. Matrices with coefficients in R_q are denoted by bold uppercase letters. In this context, implementations such as [13,17] represent $a \in R_q$ as a structure of n integers, often named `poly`, while an element $\mathbf{a} \in R_q^k$ (resp. $\mathbf{b} \in R_q^l$) is represented as a structure of k (resp. l) `poly` and is commonly named `polyveck` (resp. `polyvecl`). In the remainder, we perform polynomial operations in R_q unless otherwise noted.

For an even (resp. odd) positive integer α, we define $r_0 = r \bmod^{\pm} \alpha$ to be the unique element r' in the range $-\frac{\alpha}{2} < r' \leq \frac{\alpha}{2}$ (resp. $-\frac{\alpha-1}{2} < r' \leq \frac{\alpha-1}{2}$) such that

$r \equiv r \bmod \alpha$. For $\alpha \in \mathbb{N}$, we define $r' = r \bmod^{+} \alpha$ to be the unique element r' in the range $0 \leq r' < \alpha$.

For an element $w \in \mathbb{Z}_q$, we define $\|w\|_\infty$ as $|w \bmod^{\pm} q|$. For an element $\mathbf{w} \in R$, i.e., $\mathbf{w} = w_0 + w_1 X + \ldots + w_{n-1} X^{n-1}$, we define $\|\mathbf{w}\|_\infty$ as $\max_i \|w_i\|_\infty$ and we define $\|\mathbf{w}\| = \sqrt{\|w_0\|_\infty^2 + \|w_1\|_\infty^2 + \ldots + \|w_{n-1}\|_\infty^2}$.

Let $S_\eta = \{\mathbf{w} \in R : \|\mathbf{w}\|_\infty \leq \eta\}$ and $\tilde{S}_\eta$ the set $\{\mathbf{w} \bmod^{\pm} 2\eta : \mathbf{w} \in R\}$.

For $\lambda \in \mathbb{Z}_q$ and an element $\mathbf{h}$ of k vectors of n coefficients. We define $|\mathbf{h}|_{h_j=\lambda}$ as the total number of coefficients of $\mathbf{h}$ equal to λ.

$[\![\texttt{op}]\!]$ represents the boolean evaluation of the operation `op`.

2.1 Presentation of Dilithium

In 2022 Dilithium [3] was selected alongside Falcon [35] and SPHINCS$^+$ [4], yet it is the recommended PQC signature scheme for most use cases. Dilithium is a lattice-based signature scheme based on the Fiat-Shamir with aborts principle [22] and proposed by the "Cryptographic Suite for Algebraic Lattices" (CRYSTALS) team. Its security is derived from the hardness of solving the module learning with errors (M-LWE) [3], and SelfTargetMSIS [18] problems. For a given $(k,\, l) \in \mathbb{N}$, it operates over the module $R_q^{k\times l}$, with fixed $q = 8380417$, a 23-bit integer and $n = 256$. There are three security levels, from NIST level 1 to level 5, with changes essentially in the $(k,\, l)$ chosen for the module $R_q^{k\times l}$. There are also two variants of the signing algorithm, one deterministic and one randomized, the only difference being how the randomness is sampled. For efficiency, the scheme makes use of rounding sub-functions such as `Power2Round`, `Decompose`, `HighBits`, `LowBits`, `MakeHint`, `UseHint` whose latest specification can be found in [3]. To reduce memory storage, polynomials are stored as a byte stream, using packing and unpacking functions specified in [3].

Algorithm 1 KeyGen

```
Output: $pk = (\rho,\, \mathbf{t}_1)$, $sk = (\rho,\, K,\, tr,\, \mathbf{s}_1,\, \mathbf{s}_2,\, \mathbf{t}_0)$
1 $\zeta \leftarrow \{0,1\}^{256}$
2 $(\rho, \rho', \mathcal{K}) \in \{0,1\}^{256} \times \{0,1\}^{512} \times \{0,1\}^{256} := \mathrm{H}(\zeta)$ ▷ H instantiated as SHAKE-256
3 $\mathbf{A} \in R_q^{k\times l} := \texttt{ExpandA}(\rho)$ ▷ $\mathbf{A}$ is generated and stored in NTT Representation as $\hat{\mathbf{A}}$
4 $(\mathbf{s}_1,\, \mathbf{s}_2) \in S_\eta^l \times S_\eta^k := \texttt{ExpandS}(\rho')$
5 $\mathbf{t} := \mathbf{A}\,\mathbf{s}_1 + \mathbf{s}_2$    ▷ Compute $\mathbf{A}\mathbf{s}_1$ as $\texttt{NTT}^{-1}(\hat{\mathbf{A}}\,\texttt{NTT}(\mathbf{s}_1))$
6 $(\mathbf{t}_1,\, \mathbf{t}_0) := \texttt{Power2Round}_q(t,\, d)$
7 $tr \in \{0,1\}^{256} := \mathrm{H}(\rho \,||\, \mathbf{t}_1)$
8 return $pk = (\rho,\, \mathbf{t}_1)$, $sk = (\rho,\, \mathcal{K},\, tr,\, \mathbf{s}_1,\, \mathbf{s}_2,\, \mathbf{t}_0)$
```

Key Generation. The key generation in Algo. 1 expands a matrix $\mathbf{A} \in R_q^{k\times l}$ from a public seed ρ via a function `ExpandA` [3]. It then samples random secret vectors $\mathbf{s}_1$ and $\mathbf{s}_2$. Elements of these vectors belong to R_q with small coefficients of size at most η, a small integer. The second part of the public key, denoted pk, is computed as $\mathbf{t} = \mathbf{A}\,\mathbf{s}_1 + \mathbf{s}_2$ but, for efficiency, only the higher bits $\mathbf{t}_1$ are made public while the lower part $\mathbf{t}_0$ is kept secret. Finally, tr is the hash of the pk and is added to the secret key, sk.

Algorithm 2 Sign

```
     Input  : $sk = (\rho, \mathcal{K}, tr, \mathbf{s}_1, \mathbf{s}_2, \mathbf{t}_0)$
     Output: $\sigma = (\tilde{c}, \mathbf{z}, \mathbf{h})$
 1 $\mathbf{A} \in R_q^{k \times l} := \texttt{ExpandA}(\rho)$ ▷ $\mathbf{A}$ is generated and stored in NTT representation as $\hat{\mathbf{A}}$
 2 $\mu \in \{0,1\}^{512} := \mathrm{H}(tr \,||\, M)$
 3 $\kappa := 0,\ (\mathbf{z}, \mathbf{h}) := \perp$
 4 $\rho' \in \{0,1\}^{512} := \mathrm{H}(\mathcal{K} \,||\, \mu)$ (or $\rho' \leftarrow \{0,1\}^{512}$ for randomized signing)
 5 while $(\mathbf{z}, \mathbf{h}) = \perp$ do ▷ Pre-compute $\hat{s}_1 := \texttt{NTT}(\mathbf{s}_1)$, $\hat{s}_2 := \texttt{NTT}(\mathbf{s}_2)$ and $\hat{\mathbf{t}}_0 := \texttt{NTT}(\mathbf{t}_0)$
 6     $\mathbf{y} \in \tilde{S}_{\gamma_1}^{l} := \texttt{ExpandMask}(\rho', \kappa)$          ▷ $\kappa$ is increased by 1 at each call
 7     $\mathbf{w} := \mathbf{A}\,\mathbf{y}$                                ▷ $\mathbf{w} := \texttt{NTT}^{-1}(\hat{\mathbf{A}} \cdot \texttt{NTT}(\mathbf{y}))$
 8     $\mathbf{w}_1 = \texttt{HighBits}_q(\mathbf{w}, 2\gamma_2)$
 9     $\tilde{c} \in \{0,1\}^{256} := \mathrm{H}(\mu \,||\, \mathbf{w}_1)$
10     $c \in B_\tau := \texttt{SampleInBall}(\tilde{c})$     ▷ Store $c$ in NTT representation as $\hat{c} = \texttt{NTT}(c)$
11     $\mathbf{z} := \mathbf{y} + c\,\mathbf{s}_1$                         ▷ Compute $c\mathbf{s}_1$ as $\texttt{NTT}^{-1}(\hat{c} \cdot \hat{s}_1)$
12     $\mathbf{r}_0 := \texttt{LowBits}_q(\mathbf{w} - c\mathbf{s}_2, 2\gamma_2)$        ▷ Compute $c\mathbf{s}_2$ as $\texttt{NTT}^{-1}(\hat{c} \cdot \hat{s}_2)$
13     if $\|\mathbf{z}\|_\infty \geq \gamma_1 - \beta$ or $\|\mathbf{r}_0\|_\infty \geq \gamma_2 - \beta$ then
14         $(\mathbf{z}, \mathbf{h}) := \perp$
15     else
16         $\mathbf{h} := \texttt{MakeHint}_q(-c\,\mathbf{t}_0, \mathbf{w} - c\mathbf{s}_2 + c\,\mathbf{t}_0, 2\gamma_2)$ ▷ Compute $c\,\mathbf{t}_0$ as $\texttt{NTT}^{-1}(\hat{c} \cdot \hat{\mathbf{t}}_0)$
17         if $||c\,\mathbf{t}_0||_\infty \geq \gamma_2$ or $|\mathbf{h}|_{\mathbf{h}_j=1} > \omega$ then
18             $(\mathbf{z}, \mathbf{h}) := \perp$
19 return $\sigma = (\tilde{c}, \mathbf{z}, \mathbf{h})$
```

Signature. The signature, described in Algo. 2, consists of a rejection sampling loop, generating a new signature until it satisfies some security and correctness properties. The rejection loop starts by generating a masking vector $\mathbf{y}$[1] with coefficients below γ_1. Then, the signer computes $\mathbf{w} = \mathbf{A}\mathbf{y}$ and compresses it into high-order bits $\mathbf{w}_1$ and low-order bits $\mathbf{w}_0$. The message is hashed with $\mathbf{w}_1$ to sample a specific ternary challenge c with fixed weight τ and the rest 0's. The potential signature is computed, using $\mathbf{s}_1$ as $\mathbf{z} = \mathbf{y} + c\,\mathbf{s}_1$. Because the verifier does not know $\mathbf{t}_0$, the signature includes a vector $\mathbf{h}$ that keeps track of the coefficients that overflow onto the high part of $\mathbf{w}_1$. Checks are performed in lines 13 and 17 to ensure that no information about the secret key leaks and for correctness.
If any of these checks fails, a new signature candidate is generated.

Signature Verification. The verification algorithm, described in Algo. 3, involves computing the high-order bits of $\mathbf{A}\,\mathbf{z} - c\,\mathbf{t}_1\,2^d$ using the signature and the public key, pk. The result is then corrected by the hint vector $\mathbf{h}$. If the signature is correct, this is equal to $\mathbf{w}_1$, which allows to recompute the challenge c. To verify a signature, a final check ensures that all the coefficients of $\mathbf{z}$ are less than $\gamma_1 - \beta$ and that the number of hints in $\mathbf{h}$ are less than ω.

[1] This vector is essentially used as a random mask of the secret $\mathbf{s}_1$ line 11 in Algo. 1.

Algorithm 3 Verify

```
Input  : $pk = (\rho,\, \mathbf{t}_1)$, $\sigma = (\tilde{c}, \mathbf{z}, \mathbf{h})$
Output: $True$ or $False$
1 $\mathbf{A} \in R_q^{k \times l} := \texttt{ExpandA}(\rho)$
2 $\mu \in \{0,1\}^{512} := \mathrm{H}(\mathrm{H}(\rho \,||\, \mathbf{t}_1) \,||\, M)$
3 $c := \texttt{SampleInBall}(\tilde{c})$
4 $\mathbf{w}_1' := \texttt{UseHint}_q(\mathbf{h}, \mathbf{A}\mathbf{z} - c\mathbf{t}_1\, 2^d, 2\gamma_2)$
5 return $[\![ ||\mathbf{z}||_\infty < \gamma_1 - \beta ]\!]$ and $[\![ \tilde{c} = \mathrm{H}(\mu \,||\, \mathbf{w}_1') ]\!]$ and $[\![ |\mathbf{h}|_{\mathbf{h}_j = 1} \leq \omega ]\!]$
```

2.2 Fault Models

Over the past two decades, fault injection attacks have emerged as a powerful method of compromising devices, even secured ones [39]. These attacks use a variety of techniques, including laser beam and electromagnetic (EM) pulse, which allow precise control over space and time. Extensive research has focused on characterizing the effects of fault injection in order to identify fault models at a given abstraction (circuit level, hardware logical level, assembly code, source code). Such models serve as a framework for categorizing and studying the potential fault attacks on systems and sub-systems such as cryptographic ones.

A fault model at hardware logical level includes the width of the fault (mono-bit, multi-bits, byte, or word) and the induced change (bit set, bit reset, bit-flip, random changes). The feasibility and cost of a fault model are related to the required equipment, the time, and the level of expertise needed. At software level, faults impact the data and computations, the control flow and executed instructions, or both. At this level, common fault models cover effects such as instruction skipping or conditional branch inversion. A threat (or attacker) model defines both the fault models and the number of faults needed to study the security of a system. Note that achieving a precise fault multiple times generally requires a strong attacker and means increased difficulty in real-life scenarios.

In this paper, we consider four fault models at C level, two on instruction flow and two on data, namely skipping faults, test inversion faults, randomization faults, and zeroizing faults. For each we explain how they can be achieved.

Skipping fault involves deliberately skipping selected lines of code within the execution of a program. It amounts to not executing specific program instructions or intentionally manipulating the program counter. Skipping faults can be achieved with many fault injection techniques such as CPU clock or voltage glitching [19,36,40], EM pulse injection [24] or laser beam [14]. Skipping faults can have severe consequences, such as bypassing critical security checks or essential cryptographic steps. The practical demonstration of higher-order skipping faults has recently underscored the significance of this type of fault attack [9], highlighting the potential security risks they pose.

Test inversion fault corresponds to the inversion of a conditional branch outcome (if-then or if-then-else constructs). It can be achieved by inverting the condition in the corresponding conditional jump instruction or by corrupting an instruction involved in the condition's computation, all of which can be achieved by several injection means [20,11,26]. It can also be the result of skipping the branch instruction. Test inversion enables bypassing security verification.

Zeroizing fault assumes that the attacker can set a variable, a constant or a portion thereof, to zero. While the realism of this attack scenario has been questioned, instances of zeroizing faults have been successfully executed in practice as it amounts to resetting or flipping some bits [6]. State-of-the-art fault injection allows controlling the fault affecting up to a dozen bits [10]. Therefore, zeroing less than a dozen bits can be considered as realistic. Zeroizing faults can also be a consequence of skipping faults, or instruction or operand corruption [26,37].

Randomization fault introduces random changes to data or computations within a targeted program, causing unexpected behavior. This means that after the injection, the attacker remains unaware of the exact result of the computation, but gains an advantage by knowing that it has been altered within a specific range. Randomization faults can lead to incorrect output, bypassing security checks, or compromising the integrity of the cryptographic operation.

In the remainder, we only consider the type and number of faults for each scenarios from which one can deduce the corresponding attacker model.

2.3 Related Works

In this section, we first give an overview of the state of the art in fault attacks on Dilithium. Then, we explain how one can forge signatures with partial information about the secret key.

Fault Attacks on Dilithium For its PQC competition, NIST put an emphasis on security against side-channel and fault attacks. In this regard, Dilithium already has some constant-time properties to an extent. Still, several practical fault injection attacks leading to the key recovery have already been published. Among them, Bruinderink and Pessl [8] demonstrated the applicability of differential fault attacks on the deterministic Dilithium through multiple paths.

In contrast to the extensive research on fault injection attacks on the signature algorithm, the verification process has received less attention. The main reason is that only public information are handled during the verification process. Fault attacks on the verification procedure primarily target the comparison in line 5 of Algo. 3, which is usually carefully implemented on secure devices to prevent acceptance of corrupted signatures. The algebraic parts of the verification process are often considered less sensitive, given the difficulty of forging a signature. Nonetheless, exploitable vulnerabilities can make these algebraic parts an attractive attack surface for fault injection. It is the case for the RSA signature scheme where manipulating the modulus N, which is sensitive to faults, allows an attacker to pass the verification with false signatures [33].

Although there have been efforts to explore fault injection attacks in the context of the verification procedure, such research remains scarce.
One notable study conducted by Bindel et al. [5] highlighted the potential

consequences of zeroizing the challenge c within the verification process of other lattice-based signature schemes. They demonstrated that such zeroization could enable successful verification of invalid signatures for any message, all without needing the secret key. Furthermore, they showed that skipping the correctness check or the size check on $\mathbf{z}$ in line 5 of Algo. 3 could have the same effect.

Achieving zeroization in practice is not a trivial task, and concerns have been raised regarding the practicality of this specific theoretical attack scenario. However, recently, Ravi et al. [31] presented the first practical zeroization fault attack on an implementation of the Dilithium signature verification. They showed that zeroizing the twiddle constants, a fixed table of coefficients, in the `NTT` reduces its entropy, thus achieving the same effect as Bindel et al. who zeroized the challenge c. They practically demonstrated this by noticing that zeroizing the starting address of the the twiddle constant's table is sufficient to set them all to zero.

Given the critical role of the verification algorithm in upholding the security of digital systems, it is essential to dedicate more attention to comprehensively evaluating its susceptibility to fault injection attacks.

Dilithium Signature Forgery In the following, we present how to forge Dilithium signatures, assuming that only the $\mathbf{s}_1$ part of the secret key is known.

The verification algorithm essentially recomputes the value of $\mathbf{w}_1$ using only public information to accept a signature. To sign a message, the sk used is composed of (ρ, K, tr, $\mathbf{s}_1$, $\mathbf{s}_2$, $\mathbf{t}_0$). The seed ρ needed to expand the matrix $\mathbf{A}$ is also part of the public key and tr is the hash of pk, so they can be both retrieved from public information. The nonce $\mathcal{K}$ is used to generate the vector $\mathbf{y}$, but no check on the verification allows to determine if this particular value was used. Thus, it can be replaced by a random value. The $\mathbf{s}_2$ part of the secret key is only used for rejection checks and in intermediate values. Regarding $\mathbf{t}_0$, the security proof considers it as public [3], but in practice it remains secret.

In the signing algorithm, if we assume that $\mathbf{y}$ is randomly chosen, as in the randomized version of Dilithium, then, with the knowledge of the public key and $\mathbf{s}_1$, an attacker can proceed up to the computation of $\mathbf{z}$, in line 11 of Algo. 2. From this step, there are basically two different methods to forge a signature.

Bruinderink and Pessl [8] presented a modified signing procedure to perform signature forgery with only public information and the knowledge of $\mathbf{s}_1$. They first compute with known values the value $\mathbf{u} := \mathbf{A}\,\mathbf{s}_1 - \mathbf{t}_1\,2^d = \mathbf{t}_0 - \mathbf{s}_2$. Given $\mathbf{s}_2$'s small coefficients, the quantity $\mathbf{u}$ approximates $\mathbf{t}_0$. It allows them to compute an alternative $\mathbf{h}$ with $\mathbf{u}$. They cannot check the rejections based on $\mathbf{s}_2$ and $\mathbf{t}_0$, so they remove them because this will not impact the correctness of the signature with high probability.

Ravi et al. [30] also showed an alternative signature forgery procedure. They start in the same way as Bruinderink and Pessl up to line 11 of Algo. 2. They showed that the `UseHint` procedure can be inverted and used to compute the high bits of $\mathbf{w} - c\,\mathbf{s}_2$. But, because $\|\texttt{LowBits}_q(\mathbf{A}\,\mathbf{z} - c\,\mathbf{s}_2,\, 2\gamma_2)\|_\infty < \gamma_2 - \beta$ with probability very close to 1, they can be sure to recompute the correct $\mathbf{w}_1$.

3 Public parameters sensitivity analysis of Verify

In this section, we present the main idea allowing the acceptance of random signatures by Algo. 3 through the exploitation of specific faults. Then, we conduct a comprehensive analysis of one implementation of the verification algorithm. The goal is to identify sensitive operations and explain how to forge signatures that would be accepted, in the presence of the corresponding fault. To our knowledge, this is the first extensive study of the verification algorithm. Finally, we summarize the sensitivity of each location regarding our fault models.

3.1 Main Idea

The attacker's goal, here, is to produce a message-signature pair that will be accepted by Algo. 3. The main idea behind the signature verification of Dilithium is that computing the value $\mathbf{A}\,\mathbf{z} - c\,\mathbf{t}_1\,2^d$ will be equal to the high bits of $\mathbf{A}\,\mathbf{y}$ plus some bounded small values. These small values can sometimes slightly overflow onto the high part $\mathbf{w}_1$, so the signing algorithm computes a specific vector $\mathbf{h}$ of hints that will be used to compensate for this effect. Given this hint, one can retrieve the same $\mathbf{w}_1$ as in the signing procedure but with only public values.

Let us remember that a Dilithium signature is composed of $(c, \mathbf{z}, \mathbf{h})$, given that $\mathbf{z} = \mathbf{y} + c\,\mathbf{s}_1$ and $\mathbf{t} = \mathbf{A}\,\mathbf{s}_1 + \mathbf{s}_2$, we have

$$\mathbf{A}\,\mathbf{z} - c\,\mathbf{t} = \mathbf{A}\,\mathbf{y} - c\,\mathbf{s}_2. \tag{1}$$

By replacing $\mathbf{w} = \mathbf{A}\,\mathbf{y}$ and $\mathbf{t} = \mathbf{t}_1\,2^d + \mathbf{t}_0$[2]in Equation 1, we get

$$\mathbf{A}\,\mathbf{z} - c\,\mathbf{t}_1\,2^d - c\,\mathbf{t}_0 = \mathbf{w} - c\,\mathbf{s}_2,$$

and by rewriting this equation, we obtain

$$\mathbf{A}\,\mathbf{z} - c\,\mathbf{t}_1\,2^d = \mathbf{w} - c\,\mathbf{s}_2 + c\,\mathbf{t}_0. \tag{2}$$

Equation 2 is exactly the quantity computed for the verification line 4 of Algo. 3. Remember that $\mathbf{h} = \mathtt{MakeHint}_q(-c\,\mathbf{t}_0,\, \mathbf{w} - c\,\mathbf{s}_2 + c\,\mathbf{t}_0,\, 2\,\gamma_2)$.
Then, from Lemma 1.1 in [3], we know that

$$\mathtt{UseHint}_q(\mathbf{h},\, \mathbf{w} - c\,\mathbf{s}_2 + c\,\mathbf{t}_0,\, 2\gamma_2) = \mathtt{HighBits}_q(\mathbf{w} - c\mathbf{s}_2,\, 2\gamma_2). \tag{3}$$

Since $\|c\,\mathbf{s}_2\|_\infty \le \beta$ and $\|\mathtt{LowBits}_q(\mathbf{w} - c\mathbf{s}_2,\, 2\gamma_2)\|_\infty < \gamma_2 - \beta$[3], according to Lemma 2 [3] we have

$$\mathtt{HighBits}_q(\mathbf{w} - c\mathbf{s}_2,\, 2\gamma_2) = \mathtt{HighBits}_q(\mathbf{w}, 2\gamma_2) = \mathbf{w}_1, \tag{4}$$

which shows how to retrieve the value $\mathbf{w}_1$ from the public key and the signature.

From line 4 in Algo. 3 and the equations above, we can see that, at the top level, $\mathbf{w}_1$ is only dependant on $\mathbf{A}$, $\mathbf{z}$, c, $\mathbf{t}_1$ and $\mathbf{h}$ which are known. The matrix $\mathbf{A}$ can be considered as a fixed value, like the constant d. The values $\mathbf{z}$ and $\mathbf{h}$ are essentially random values on their respective intervals, and an attacker can choose them freely.

[2] The vector $\mathbf{w}$ is computed line 7 in Algo. 2 and $\mathbf{t}$ is computed line 5 in Algo. 1.

[3] If σ is a valid signature then we know that this condition is fulfilled thanks to the check on $\mathbf{r}_0$ on line 13 of Algo. 2.

Given this information, we show how to bound the value $c\,\mathbf{t}_1\,2^d$ so that it doesn't impact too much the high bits of $\mathbf{A}\,\mathbf{z}$.

Proposition 1. *Let $\boldsymbol{z} \in \tilde{S}^l_{\gamma_1-\beta}$ be a random vector. If at least one of the following conditions is satisfied:*

P1. $c\,\boldsymbol{t}_1\,2^d = 0$

P2. $\|c\,\boldsymbol{t}_1\,2^d\|_\infty \leq \beta$ *and* $\|\mathtt{LowBits}_q(\boldsymbol{A}\,\boldsymbol{z} - c\,\boldsymbol{t}_1\,2^d,\, 2\gamma_2)\|_\infty < \gamma_2 - \beta$

P3. $\|c\,\boldsymbol{t}_1\,2^d\|_\infty \leq \gamma_2$ *and* $\boldsymbol{h}' = \mathtt{MakeHint}_q(c\,\boldsymbol{t}_1\,2^d,\, \boldsymbol{A}\,\boldsymbol{z} - c\,\boldsymbol{t}_1\,2^d,\, 2\,\gamma_2)$

Then, $\mathtt{HighBits}_q(\boldsymbol{A}\,\boldsymbol{z} - c\,\boldsymbol{t}_1\,2^d,\, 2\gamma_2) = \mathtt{HighBits}_q(\boldsymbol{A}\,\boldsymbol{z}, 2\gamma_2)$.

Proof.

P1. If $c\,\mathbf{t}_1\,2^d = 0$, the result is straightforward.

P2. Direct application of Lemma 2 in [3].

P3. If $\|c\,\mathbf{t}_1\,2^d\|_\infty \leq \gamma_2$ then from Lemma 1.1 in [3], we know that $\mathtt{UseHint}_q\big(\mathtt{MakeHint}_q(c\,\mathbf{t}_1\,2^d,\, \mathbf{A}\,\mathbf{z} - c\,\mathbf{t}_1\,2^d,\, 2\,\gamma_2),\, \mathbf{A}\,\mathbf{z} - c\,\mathbf{t}_1\,2^d,\, 2\,\gamma_2\big)$
$= \mathtt{HighBits}_q(\mathbf{A}\,\mathbf{z} - c\,\mathbf{t}_1\,2^d + c\,\mathbf{t}_1\,2^d,\, 2\gamma_2)$. □

If we can fault some operations of Algo. 3 and have one of these three conditions, then we can carefully construct signatures that will pass the verification.

– Even though at first glance P1 seems like a strong hypothesis to realize, Ravi et al. [31] recently showed the practical realization of a fault attack involving the challenge c that has the same effect.
– P2 is perhaps the hardest hypothesis to use because we need both conditions for the fault to have the desired effect.
– P3 seems convenient because the hint vector $\mathbf{h}$ is part of the signature, and γ_2 is not too small.

To illustrate the sensitivity analysis, we use the C implementation of Dilithium from the PQclean library [17], which is identical to the reference one [13] but more portable. The code structure is also reused in other implementations [16]. The function `PQCLEAN_DILITHIUM2_CLEAN_crypto_sign_verify` will be referred to as `verify` to simplify notations and is given in Fig. 1. In the following, we describe three relevant scenarios resulting from the analysis of the C code. For each scenario, we identify which fault model, as presented in Sec. 2.2, allows us to exploit propositions P1 and P3 to forge a signature. We provide two algorithms Algo. 4 for P1 and Algo. 5 for P3, to forge signatures given the corresponding faults. Each algorithm has been implemented in SageMath. We have verified that such carefully forged signatures using these algorithms, paired with specific fault effects, enable us to pass the verification.

3.2 Preliminary Analysis

A natural target is the corruption of the value returned by the verification process. An attacker must then force the return value to 0, corresponding to a valid signature. However, zeroizing 32 bits may be relatively hard for an attacker to accomplish in practice. Alternatively, the attacker can try to pass all of the three checks lines 14, 16, and 54 of `verify` in Fig. 1, necessitating three test

```c
int verify(const uint8_t *sig, size_t siglen, const uint8_t *m, size_t mlen,
           const uint8_t *pk) {
    unsigned int i;
    uint8_t buf[K * POLYW1_PACKEDBYTES], rho[SEEDBYTES], mu[CRHBYTES];
    uint8_t c[SEEDBYTES], c2[SEEDBYTES];
    poly cp;
    polyvecl mat[K], z;
    polyveck t1, w1, h;
    shake256incctx state;
    if (siglen != CRYPTO_BYTES)
        return -1;

    unpack_pk(rho, &t1, pk);
    if (unpack_sig(c, &z, &h, sig))
        return -1;
    if (polyvecl_chknorm(&z, GAMMA1 - BETA))
        return -1;

    /* Compute CRH(H(rho, t1), msg) */
    shake256(mu, SEEDBYTES, pk, CRYPTO_PUBLICKEYBYTES);
    shake256_init(&state);
    shake256_absorb(&state, mu, SEEDBYTES);
    shake256_absorb(&state, m, mlen);
    shake256_finalize(&state);
    shake256_squeeze(mu, CRHBYTES, &state);

    /* Matrix-vector multiplication; compute Az - c2^dt1 */
    poly_challenge(&cp, c);
    polyvec_matrix_expand(mat, rho);

    polyvecl_ntt(&z);
    polyvec_matrix_pointwise_montgomery(&w1, mat, &z);

    poly_ntt(&cp);                     Scenario 1: Sampling of c̃
    polyveck_shiftl(&t1);              Scenario 2: Shift by d
    polyveck_ntt(&t1);
    polyveck_pointwise_poly_montgomery(&t1, &cp, &t1);

    polyveck_sub(&w1, &w1, &t1);   Scenario 3: Subtraction
    polyveck_reduce(&w1);
    polyveck_invntt_tomont(&w1);

    /* Reconstruct w1 */
    polyveck_caddq(&w1);
    polyveck_use_hint(&w1, &w1, &h);
    polyveck_pack_w1(buf, &w1);

    /* Call random oracle and verify challenge */
    shake256_init(&state);
    shake256_absorb(&state, mu, CRHBYTES);
    shake256_absorb(&state, buf, K * POLYW1_PACKEDBYTES);
    shake256_finalize(&state);
    shake256_squeeze(c2, SEEDBYTES, &state);
    for (i = 0; i < SEEDBYTES; ++i) {
        if (c[i] != c2[i]) {
            return -1;
        }
    }
    return 0;
}
```

Fig. 1: PQClean Dilithium `verify` code snippet.

inversion faults at minimum. These sensitive tests are typically hardened in secure applications [38], making such fault effects potentially hard to achieve. The analysis below focuses on arithmetic parts that might be less carefully implemented since they do not handle secure parameters.

```
 1 void unpack_pk(uint8_t rho[SEEDBYTES], polyveck *t1,
 2                const uint8_t pk[CRYPTO_PUBLICKEYBYTES]) {
 3   unsigned int i;
 4   for (i = 0; i < SEEDBYTES; ++i) {
 5     rho[i] = pk[i];
 6   }
 7   pk += SEEDBYTES;
 8   for (i = 0; i < K; ++i) {
 9     polyt1_unpack(&t1->vec[i], pk + i * POLYT1_PACKEDBYTES);
10   }
11 }
```

Fig. 2: PQClean unpack *pk* code snippet

```
1 void polyt1_unpack(poly *r, const uint8_t *a) {
2   unsigned int i;
3   for (i = 0; i < N / 4; ++i) {
4     r->coeffs[4*i + 0] = ((a[5*i + 0] >> 0) | ((uint32_t)a[5*i + 1] << 8)) & 0x3FF;
5     r->coeffs[4*i + 1] = ((a[5*i + 1] >> 2) | ((uint32_t)a[5*i + 2] << 6)) & 0x3FF;
6     r->coeffs[4*i + 2] = ((a[5*i + 2] >> 4) | ((uint32_t)a[5*i + 3] << 4)) & 0x3FF;
7     r->coeffs[4*i + 3] = ((a[5*i + 3] >> 6) | ((uint32_t)a[5*i + 4] << 2)) & 0x3FF;
8   }
9 }
```

Fig. 3: PQClean unpack $\mathbf{t}_1$ code snippet

For instance, the unpacking of $\mathbf{t}_1$ is a potential location for fault injection. To avoid affecting other public variables, such as $\mathbf{A}$, the only feasible target is the constant `0x3FF` lines 4 to 7 of the function `polyt1_unpack` Fig. 3. Zeroizing this constant sets every coefficient of $\mathbf{t}_1$ to zero and we can use P1 through Algo. 4, detailed in Scenario 1. However, this approach requires a total of $K \times N$ repeated faults, which can be challenging in practice. Yet, it is worth noting that $\mathbf{t}_1$ could be sensitive if declared as a global variable. Then, as by default it is initialized to 0, faulting the call to the function `polyt1_unpack`, line 9 in Fig. 2, could set $\mathbf{t}_1$ to 0 with just K repeated faults. Alternatively, one test inversion fault, line 8 Fig. 2, can force zero iterations of the loop.

```
void invntt_tomont(int32_t a[N]) {
  unsigned int start, len, j, k;
  int32_t t, zeta;
  const int32_t f = 41978; // mont^2/256
  k = 256;
  for (len = 1; len < N; len <<= 1) {
    for (start = 0; start < N; start = j + len) {
      zeta = -zetas[--k];
      for (j = start; j < start + len; ++j) {
        t = a[j];
        a[j] = t + a[j + len];
        a[j + len] = t - a[j + len];
        a[j + len] = montgomery_reduce((int64_t)zeta * a[j + len]);
      }
    }
  }
  for (j = 0; j < N; ++j)
    a[j] = montgomery_reduce((int64_t)f * a[j]);
}
```

Fig. 4: PQClean $\texttt{NTT}^{-1}$ code snippet

Our attention also turns to lines 34, 36, and 41 of Fig. 1 involving the `NTT` and $\texttt{NTT}^{-1}$ conversions, given in Fig. 4 . Notably, Ravi et al. [31] already cover the conversion of c in line 34. At the end of the inverse conversion of $\mathbf{A}\,\mathbf{z} - c\,\mathbf{t}_1\,2^d$ each coefficient undergoes multiplication by the squared Montgomery factor divided by 256 in a `for` loop, line 18 Fig. 4. This 32-bit integer constant plays a critical role. It is used at each of the N iterations so it can potentially be stored in a register. Zeroizing this value once can set all polynomial $\mathbf{A}\,\mathbf{z} - c\,\mathbf{t}_1\,2^d$ to 0. However, this fault must be repeated K times, once for each polynomial of the vector processed by the $\texttt{NTT}^{-1}$. We can exploit this fault to sample the challenge c with $\mathbf{w}_1 = 0$ and forge valid signatures with Algo. 4. We notice that even if we first perform the $\texttt{NTT}^{-1}$ of $\mathbf{A}\,\mathbf{z}$ and $c\,\mathbf{t}_1\,2^d$ separately, and then subtract the two, it would also be vulnerable. This is because we can apply the same fault to the $\texttt{NTT}^{-1}$ of $c\,\mathbf{t}_1\,2^d$ to zeroize the result, enabling the exploitation of P1.

3.3 Scenario 1: Sampling of $\tilde{c}$

```
1  void poly_challenge(poly *c, const uint8_t seed[SEEDBYTES]) {
2    unsigned int i, b, pos;
3    uint64_t signs;
4    uint8_t buf[SHAKE256_RATE];
5    shake256incctx state;
6    shake256_init(&state);
7    shake256_absorb(&state, seed, SEEDBYTES);
8    shake256_finalize(&state);
9    shake256_squeeze(buf, sizeof buf, &state);
10   signs = 0;
11   for (i = 0; i < 8; ++i)
12     signs |= (uint64_t)buf[i] << 8 * i;
13   pos = 8;
14   for (i = 0; i < N; ++i)
15     c->coeffs[i] = 0;
16   for (i = N - TAU; i < N; ++i) {
17     do {
18       if (pos >= SHAKE256_RATE) {
19         shake256_squeeze(buf, sizeof buf, &state);
20         pos = 0;
21       }
22       b = buf[pos++];
23     } while (b > i);
24     c->coeffs[i] = c->coeffs[b];
25     c->coeffs[b] = 1 - 2 * (signs & 1);
26     signs >>= 1;
27   }
28   shake256_release(&state);
29 }
```

Fig. 5: PQClean sampling of c code snippet

For efficiency, the verification algorithm only compares the recomputed seed $\tilde{c}$ with the one from the signature, line 54 Fig. 1. In our investigation, we identify the procedure, in Fig. 5, for sampling the challenge c from its seed $\tilde{c}$ as sensitive. This process involves setting all N coefficients of the challenge to zero using a first `for` loop, followed by another `for` loop setting τ coefficients as 1 or -1.

By exploiting skipping or test-inversion faults, an attacker can target the `for` loop, line 16 Fig. 5, abort it prematurely, and zeroize all coefficients of c with just one correctly targeted fault.

Similarly, the same effect can be achieved by faulting the loop's termination condition, such as zeroizing the constant `TAU`.

Suppose the challenge c has been successfully manipulated to be zero. We present an algorithm enabling an attacker to exploit this effect, resulting in the acceptance of false signatures without needing the secret key.

Algorithm 4 Sign based on P1

```
Input  : $pk = (\rho, \mathbf{t}_1)$
Output: $\sigma = (\tilde{c}, \mathbf{z}, \mathbf{h})$
1 $\mathbf{A} \in R_q^{k \times l}$ := ExpandA($\rho$)    ▷ $\mathbf{A}$ is generated and stored in NTT representation $\hat{\mathbf{A}}$
2 $\mu \in \{0,1\}^{512} := \mathrm{H}(\mathrm{H}(\rho \,||\, \mathbf{t}_1) \,||\, M)$
3 $\mathbf{z} \in \tilde{S}^l_{\gamma_1 - \beta}$
4 $\mathbf{w} := \mathbf{A}\,\mathbf{z}$
5 $\mathbf{h}$ := SampleInBall$_\omega$()
6 $\mathbf{w}_1$ = UseHint$_q$($\mathbf{h}$, $\mathbf{w}$, $2\,\gamma_2$)
7 $\tilde{c} \in \{0,1\}^{256} := \mathrm{H}(\mu \,||\, \mathbf{w}_1)$
8 return $\sigma = (\tilde{c}, \mathbf{z}, \mathbf{h})$
```

Algo. 4 utilizes the fact that if $c = 0$, then $c\,\mathbf{t}_1\,2^d = 0$, therefore leveraging P1. We begin by sampling the vector $\mathbf{z}$ within the appropriate range. Similar to [31], our algorithm generates a random $\mathbf{h}$ satisfying its corresponding condition. Using the `UseHint` function, we compute the corresponding $\mathbf{w}_1$ to sample the resulting $\tilde{c}$. As observed earlier, we exploit faults that set c to 0 in the verification algorithm, meaning that the same seed $\tilde{c}$ is sampled as in Algo. 4. Unlike [31], we don't perform a rejection on the first coefficient of c because the fault in the verification does not use this condition.
As a variation of Algo. 4, we can directly set $\mathbf{h}$ to zero and use only the high bits of $\mathbf{A}\,\mathbf{z}$ to derive the seed $\tilde{c}$. It is worth noting that while $\mathbf{h}$ being completely null is a situation that could arise in practice, its probability is negligible. In current versions of Dilithium, this check is neither specified nor implemented. A thorough analysis is required to determine if adding the $\mathbf{h} = 0$ check to the verification algorithm would reject valid signatures. Furthermore, this scenario relies on the ability to set all coefficients of c to zero. Whereas, the challenge c should have precisely τ coefficients equal to 1 or -1. However, there are no checks in place to verify this in practice.

3.4 Scenario 2: Shift by d

```c
void polyveck_shiftl(polyveck *v) {
  unsigned int i;
  for (i = 0; i <K; ++i)
    poly_shiftl(&v->vec[i]);
}
```

Fig. 6: PQClean `polyvec` shift code snippet

```c
void poly_shiftl(poly *v) {
  unsigned int i;
  for (i = 0; i <N; ++i)
    a->coeffs[i] <<= D;
}
```

Fig. 7: PQClean `poly` code snippet

In this scenario, we focus on line 35 of `verify` given in Fig. 1. At this point, $\mathbf{t}_1$ has been unpacked, and the challenge c has been sampled from the seed $\tilde{c}$. Faulting either the shift of $\mathbf{t}_1$ by d or the multiplication of c with $\mathbf{t}_1$ can influence the magnitude of the product $c\,\mathbf{t}_1\,2^d$. It is important to note that the result of the multiplication of c with $\mathbf{t}_1\,2^d$, stored in the same location as $\mathbf{t}_1\,2^d$, already contains coefficients outside the exploitable range of Proposition 1. Thus, faulting this operation does not yield usable outcomes.

Now, let us analyze the multiplication of $\mathbf{t}_1$ by 2^d. By considering skipping faults, an attacker can target the call to the `polyveck_shiftl` function on line 35 of `verify` by skipping the corresponding jump instruction with one fault.

Another potential target is line 3 of Fig. 6, where faulting the loop counter terminates the function prematurely. Alternatively, the call to `poly_shiftl` on line 4 can be targeted during each of the K iterations. However, this approach requires K repeated faults and can be more challenging to achieve.

The loop line 3 of Fig. 7 can be a potential target for a single fault. Similarly, we can target line 4 but this approach also requires K repeated faults.

Regarding zeroization faults, the constant d can be targeted to zeroize a bit or a byte of its value. It is worth noting that, in practice, for all versions of Dilithium, $d = 13 =$ `0b1101`, which is 3 bits to set to zero.

Considering randomization faults on d, the difference is that this time there is no control over the value d' so most of the random faults are not usable.

Our aim is to determine the suitable d' such that $\|c\,\mathbf{t}_1\,2^{d'}\|_\infty \leq \gamma_2$ which allows us to utilize P3. Let us compute such a d' by bounding the product

$$\|c\,\mathbf{t}_1\,2^{d'}\|_\infty \leq 2^{d'}\|c\|_1\,\|\mathbf{t}_1\|_\infty, \tag{5}$$

since $\|c\|_1 = \tau$ and $\|\mathbf{t}_1\|_\infty \leq 2^{10} - 1$[4]

$$\leq 2^{d'}\,\tau\,(2^{10} - 1).$$

We want $2^{d'}\,\tau\,(2^{10}-1) \leq \gamma_2$. Therefore $d' \leq \log_2\left(\frac{\gamma_2}{\tau\,(2^{10}-1)}\right)$.

Example: For Dilithium-2 we have $d' = 1$, while for Dilithium-3 and 5 we have $d' = 2$. In practice, however, the maximum erroneous d' tolerated for any version is 3. This is explained by the fact that we have analyzed the worst possible case, and so in practice the bound can be tightened.

Algorithm 5 Sign based on P3

```
Input  : $pk = (\rho, \mathbf{t}_1)$
Output: $\sigma = (\tilde{c}, \mathbf{z}, \mathbf{h})$
1  $\mathbf{A} \in R_q^{k\times l} :=$ ExpandA($\rho$)    ▷ $\mathbf{A}$ is generated and stored in NTT representation $\hat{\mathbf{A}}$
2  $\mu \in \{0,1\}^{512} := \mathrm{H}(\mathrm{H}(\rho\,||\,\mathbf{t}_1)\,||\,M)$, $(\mathbf{h}) :=\perp$
3  while $(\mathbf{h}) =\perp$ do
4      $\mathbf{z} \in \tilde{S}^l_{\gamma_1-\beta}$
5      $\mathbf{w} := \mathbf{A}\,\mathbf{z}$
6      $\mathbf{w}_1 = \mathtt{HighBits}_q(\mathbf{w}, 2\,\gamma_2)$
7      $\tilde{c} \in \{0,1\}^{256} := \mathrm{H}(\mu\,||\,\mathbf{w}_1)$
8      $c \in B_\tau := \mathtt{SampleInBall}(\tilde{c})$
9      $\mathbf{h} := \mathtt{MakeHint}_q(-c\,\mathbf{t}_1\,2^{d'}, \mathbf{w} + c\,\mathbf{t}_1\,2^{d'}, 2\,\gamma_2)$
10     if $|\mathbf{h}|_{\mathbf{h}_j=1} > \omega$ then
11         $(\mathbf{h}) :=\perp$
12 return $\sigma = (\tilde{c}, \mathbf{z}, \mathbf{h})$
```

[4] We must have this condition fulfilled in **Sign** for a signature to be valid.

Assuming we have effectively manipulated $\mathbf{t}_1\, 2^d$ so that $\|c\,\mathbf{t}_1\, 2^{d'}\|_\infty \leq \gamma_2$, we present an algorithm, Algo. 5 enabling an attacker to exploit this with P3 and achieve the acceptance of false signatures without requiring the secret key.

Algo. 5 closely resembles the correct signing algorithm employed in Dilithium, although lacking some rejection checks that we can't verify. It operates with the vector $\mathbf{z}$ sampled within the appropriate range and leverages the hint vector computed using $c\,\mathbf{t}_1 2^d$. Using P3, supposing we managed to produce the corresponding fault, we can assure that $c\,\mathbf{t}_1\, 2^d$ remains sufficiently small to prevent excessive overflow into the higher bits. However, we still need to keep the rejection criterion based on the maximum value of non-zero coefficients within $\mathbf{h}$ for successful verification of such signatures. Our practical implementation of this algorithm, using SageMath library, has demonstrated low rejection rate for every security level of Dilithium, with no more than 3 on average.

3.5 Scenario 3: Subtraction

```
1 void polyveck_sub(polyveck *w, const polyveck *u, const polyveck *v) {
2   unsigned int i;
3   for (i = 0; i <K; ++i)
4     poly_sub(&w->vec[i], &u->vec[i], &v->vec[i]);
5 }
```

Fig. 8: PQClean `polyveck_sub` code snippet

```
1 void poly_sub(poly *c, const poly *a, const poly *b) {
2   unsigned int i;
3   for (i = 0; i <N; ++i) {
4     c->coeffs[i] = a->coeffs[i] - b->coeffs[i];
5 }
```

Fig. 9: PQClean `poly_sub` code snippet

To conclude our analysis, we direct our attention to line 39 of `verify` in Fig. 1. Notably, in current implementations, the result of the subtraction of $\mathbf{A}\,\mathbf{z}$ by $c\,\mathbf{t}_1\, 2^d$ is stored in the same variable as $\mathbf{A}\,\mathbf{z}$. Introducing a fault in the subtraction, allows us to exploit this observation and leverage P1.

First, one can skip the call to the function `polyveck_sub` on line 39 of `verify`, Fig. 1, to fault the subtraction. Similarly, line 3 of Fig. 8 can be targeted to exit the `for` loop early. Since the result is stored in the same location as the first operand, skipping the call to `poly_sub` on line 4 of Fig. 8 at each of the K iterations yields the same outcome. However, this approach necessitates K repeated faults, which can be harder to do.

Within the `poly_sub` function given in Fig. 9, we can focus on skipping the loop on line 3. Alternatively, we can target line 4 of Fig. 9, although this requires $K \times N$ repeated faults.

In this scenario, once we achieved to fault the subtraction, we leverage P1 and Algo. 4 remains applicable. It allows an attacker to produce a valid message-signature pair for verification. It is important to note that targeting this location has the same outcome as zeroizing the $\mathbf{t}_1$or zeroizing the challenge c in Scenario 1.

3.6 Experimental validation

Our primary objective is to evaluate the functionality of Algo. 4 and Algo. 5 under the conditions specified by P1 and P3, respectively. To achieve this, we have chosen to model faults exclusively at the algorithmic level. This decision is based on the following reasons:

- Within the C code, there are multiple potential locations and various types of exploitable faults that can lead to the three scenarios discussed in sections 3.3, 3.4, and 3.5.
- As outlined in Sec. 2.2, there are numerous ways to achieve the desired outcomes.
- The specific faults required will depend heavily on the target platform and binary code, which depends on the source code, and both the compiler and compilation options used.

Therefore, to cover a broad range of possible faults, we have developed three modified versions of Dilithium in Python that correspond to each scenario, and ensure the desired algorithmic effects.

- Version 1 for Scenario 1, where we arbitrarily set c to 0.
- Version 2 for Scenario 2, where we set d to match the value of d'.
- Version 3 for Scenario 3, where we removed the subtraction operation entirely.

We have validated that the signatures generated by Algo. 4 are accepted when using the versions 1 and 3. Likewise, we have verified that the signatures generated by Algo. 5 are accepted when using version 2.

4 Countermeasures

It is essential to implement the scheme thoughtfully, to minimize potential attacks, identifying and securing vulnerable operations within it. We outline several countermeasures to address the sensitive locations identified in this section.

For example, line 39 of `verify`, storing the result of $\mathbf{A}\,\mathbf{z}$ minus $c\,\mathbf{t}_1\,2^d$ in the same memory location as $c\,\mathbf{t}_1\,2^d$ prevents the exploitation of this subtraction in Scenario 3. Even if an attacker attempts to fault the subtraction, the subsequent computation of the high bits of $c\,\mathbf{t}_1\,2^d$ at line 45 of `verify` renders them unusable for accepting false signatures. Thus protecting this location with no extra cost.

Proposition P1 relies on the fact that all $K \times N$ coefficients of $c\,\mathbf{t}_1\,2^d$ are smaller than they should be. Therefore, if we can prevent even a single coefficient from being changed in size, the presented scenarios will not work.

A first set of commonly used countermeasures aims to make it more difficult for the attacker to induce faults or reproduce them [38]. There are also mechanisms that can detect and prevent fault injections targeting loops [28]. This can ensure data is handled correctly throughout the process. However, these countermeasures are fragile and complex to deploy, as we must ensure their presence in the final code.

Consequently, it is more advantageous to have a Dilithium verification algorithm that is intrinsically resistant to propositions P1 and P3. Let us introduce specific countermeasures tailored for the identified sensitive operations.

Distribution Check of the value $c\,\mathbf{t}_1\,2^d$ before the subtraction. By verifying if it is the expected one, we can effectively detect the faults used in Scenario 1 and 2. However, in practice, this means computing some statistical test on the values which can be computationally expensive

Verify d. Alternatively, we can check the correctness of the value d before using it. One way to do this verification is by first noticing that $(2^d)^{-1} = 1 - 2^{10} \mod q$, which can be computed easily and only with shift operations. Therefore, checking that $2^d \times (2^d)^{-1} = 1 \mod q$ before using the value d could ensure that it is the correct one used. However, this method only detects the faults of Scenario 2.

Split d. Another equivalent implementation would be to do the multiplication by 2^d in two times, with little overhead. If we set $d_1 > 3$ and $d_2 > 3$ such that $d = d_1 + d_2$, we can ensure that even if we fault one of the intermediate d, the result will be too big to use P3 in Scenario 2.

Alternative implementation. We can remark that by computing $\mathbf{z}' := \mathbf{z}\,(2^d)^{-1}$, at the beginning of the verification, we can write $\mathbf{A}\,\mathbf{z} - c\,\mathbf{t}_1\,2^d = \left(\mathbf{A}\,\mathbf{z}\,(2^d)^{-1} - c\,\mathbf{t}_1\right)2^d$. This time, the signatures will always be invalid if an attacker can skip the multiplication by $(2^d)^{-1}$ or by 2^d thus completely preventing Scenario 2. We give in Algo. 6 a possible implementation of this countermeasure.

Algorithm 6 Verify Alternative

```
Input  : $pk = (\rho, \mathbf{t}_1)$, $\sigma = (\tilde{c}, \mathbf{z}, \mathbf{h})$
Output: $True$ or $False$
1 $\mathbf{A} \in R_q^{k \times l} := \texttt{ExpandA}(\rho)$
2 $\mu \in \{0,1\}^{512} := \mathrm{H}(\mathrm{H}(\rho \,||\, \mathbf{t}_1) \,||\, M)$
3 $c := \texttt{SampleInBall}(\tilde{c})$
4 $\mathbf{z}' := \mathbf{z}\,(2^d)^{-1}$
5 $temp_1 := \mathbf{A}\,\mathbf{z}'$
6 $temp_2 := -c\mathbf{t}_1$
7 $temp_2 := temp_2 + temp_1$
8 $\mathbf{w}_1' := \texttt{UseHint}_q(\mathbf{h}, temp_2\,2^d,\, 2\gamma_2)$
9 return $[\![ ||\mathbf{z}||_\infty < \gamma_1 - \beta ]\!]$ and $[\![ \tilde{c} = \mathrm{H}(\mu \,||\, \mathbf{w}_1') ]\!]$ and $[\![ |\mathbf{h}|_{\mathbf{h}_j=1} \leq \omega ]\!]$
```

Norm Check. One last possible countermeasure would be to only accept a signature as valid if the check $\|c\,\mathbf{t}_1\,2^d\|_\infty > \gamma_2$ passes. The idea behind this check we introduce is that all three possibilities for Proposition 1 are based on the fact that $c\,\mathbf{t}_1\,2^d$ is smaller than it should be. By verifying if it is not too small, one can completely prevent its use. One thing to note is that the probability for every of the $K \times N$ coefficients to be naturally less than γ_2 is negligible. Thus, it should not change the verification algorithm of Dilithium. If this check doesn't affect the verification, it could prevent the faults used in Scenario 1 and 2.

Here, we give a summary of the previous two sections in the form of a table with the different scenarios, the type of fault that can be exploited for each, and the countermeasure associated.

Versions		Skipping	Test-Inv	Randomization	Zeroizing	Countermeasures
Scenario 1	for	✓	✓	-	✓	Distribution Check, Norm Check
	`TAU`	-	-	✓	✓	
Scenario 2	polyvec for	✓	✓	-	✓	Distribution Check, Norm Check, Verify d, Split d
	poly for	✓	✓	-	✓	
	d	✓	-	✓	✓	
Scenario 3	polyvec for	✓	✓	-	✓	Alternative implementation
	poly for	✓	✓	-	✓	
	function call	✓	-	-	✓	

Table 1: Summary of the vulnerable locations of the verification algorithm to the corresponding fault models. (✓: easy exploitation, ✓: possible exploitable, -: not applicable), together with the applicable countermeasures

5 Conclusion

This works aims at proving that, similarly to RSA, Dilithium verification shall be implemented carefully even if it does not handle secret data. Hence, we presented a comprehensive analysis of the verification algorithm of Dilithium, focusing on a common implementation in C and considering four common fault models: skipping faults, test inversion faults, randomization faults, and zeroizing faults. For each of them we establish a methodology for forging Dilithium signatures based on the specific type of fault employed during the verification process. Furthermore, our analysis provides valuable insights into the vulnerabilities and sensitive operations within the Dilithium verification algorithm. Building upon these findings, we propose a set of novel countermeasures covering the various scenarios introduced, and designed to mitigate the risks associated with these sensitive operations.

References

1. Alagic, G., Alperin-Sheriff, J., Apon, D., Cooper, D., Dang, Q., Miller, C., Moody, D., Peralta, R., Perlner, R., Robinson, A., Smith-Tone, D., Liu, Y.K.: Status report on the third round of the NIST post-quantum cryptography standardization process (2022)
2. Azouaoui, M., Bronchain, O., Cassiers, G., Hoffmann, C., Kuzovkova, Y., Renes, J., Schneider, T., Schönauer, M., Standaert, F.X., van Vredendaal, C.: Protecting dilithium against leakage: Revisited sensitivity analysis and improved implementations. In: CHES (2023)
3. Bai, S., Ducas, L., Kiltz, E., Lepoint, T., Lyubashevsky, V., Schwabe, P., Seiler, G., Stehlé, D.: CRYSTALS – Dilithium: Digital signatures from module lattices

4. Bernstein, D., Hülsing, A., Kölbl, S., Niederhagen, R., Rijneveld, J., Schwabe, P.: The SPHINCS+ signature framework. In: CCS (2019)
5. Bindel, N., Buchmann, J., Krämer, J.: Lattice-based signature schemes and their sensitivity to fault attacks. In: FDTC (2016)
6. Breier, J., Hou, X.: How practical are fault injection attacks, really? IEEE Access **10**, 113122–113130 (2022)
7. Brier, E., Chevallier-Mames, B., Ciet, M., Clavier, C.: Why one should also secure RSA public key elements. In: CHES (2006)
8. Bruinderink, L.G., Pessl, P.: Differential fault attacks on deterministic lattice signatures. CHES **2018**(3), 21–43 (Aug 2018)
9. Claudepierre, L., Péneau, P., Hardy, D., Rohou, E.: TRAITOR: A low-cost evaluation platform for multifault injection. In: ASSS (2021)
10. Colombier, B., Bossuet, L., Grandamme, P., Vernay, J., Chanavat, E., de Laulanié, L., Chassagne, B.: Multi-spot Laser Fault Injection Setup: New Possibilities for Fault Injection Attacks. In: CARDIS (2021)
11. Colombier, B., Menu, A., Dutertre, J., Moëllic, P., Rigaud, J., Danger, J.: Laser-induced single-bit faults in flash memory: Instructions corruption on a 32-bit microcontroller. In: IEEE HOST (2019)
12. Diffie, W., Hellman, M.: New directions in cryptography. IEEE Trans. Inf. Theory **22**(6), 644–654 (1976)
13. Ducas, L., Kiltz, E., Lepoint, T., Lyubashevsky, V., Seiler, G., Schwabe, P., Stehlé, D.: PQ-CRYSTALS, Dilithium (2022), gitHub repository. Accessed: 2022-12-15
14. Dutertre, J., Riom, T., Potin, O., Rigaud, J.: Experimental analysis of the laser-induced instruction skip fault model. In: NordSec (2019)
15. Islam, S., Mus, K., Singh, R., Schaumont, P., Sunar, B.: Signature correction attack on dilithium signature scheme. In: EuroS&P (2022)
16. Kannwischer, M., Petri, R., Rijneveld, J., Schwabe, P., Stoffelen, K.: PQM4: Post-quantum crypto library for the ARM Cortex-M4, accessed: 2022-12-15
17. Kannwischer, M., Schwabe, P., Stebila, D., Wiggers, T.: Pqclean
18. Kiltz, E., Lyubashevsky, V., Schaffner, C.: A concrete treatment of fiat-shamir signatures in the quantum random-oracle model. In: EUROCRYPT (2018)
19. Korak, T., Hoefler, M.: On the effects of clock and power supply tampering on two microcontroller platforms. In: FDTC (2014)
20. Kumar, D., Beckers, A., Balasch, J., Gierlichs, B., Verbauwhede, I.: An in-depth and black-box characterization of the effects of laser pulses on atmega328p. In: CARDIS (2019)
21. Liu, Y., Zhou, Y., Sun, S., Wang, T., Zhang, R., Ming, J.: On the security of lattice-based Fiat-Shamir signatures in the presence of randomness leakage. IEEE Trans. Inf. Forensics Secur. **16** (2021)
22. Lyubashevsky, V.: Fiat-shamir with aborts: Applications to lattice and factoring-based signatures. In: ASIACRYPT (2009)
23. Marzougui, S., Ulitzsch, V., Tibouchi, M., Seifert, J.: Profiling side-channel attacks on dilithium: A small bit-fiddling leak breaks it all. ePrint (2022)
24. Menu, A., Dutertre, J., Potin, O., Rigaud, J., Danger, J.: Experimental analysis of the electromagnetic instruction skip fault model. In: DTIS (2020)
25. Migliore, V., Gérard, B., Tibouchi, M., Fouque, P.A.: Masking dilithium. In: ACNS (2019)
26. Moro, N., Dehbaoui, A., Heydemann, K., Robisson, B., Encrenaz, E.: Electromagnetic fault injection: Towards a fault model on a 32-bit microcontroller. In: FDTC (2013)

27. Muir, A.: Seifert's RSA fault attack: Simplified analysis and generalizations
28. Proy, J., Heydemann, K., Berzati, A., Cohen, A.: Compiler-assisted loop hardening against fault attacks. ACM 2017 (2017)
29. Qiao, Z., Liu, Y., Zhou, Y., Ming, J., Jin, C., Li, H.: Practical public template attack attacks on CRYSTALS-Dilithium with randomness leakages. IEEE Trans. Inf. Forensics Secur. (2023)
30. Ravi, P., Jhanwar, M.P., Howe, J., Chattopadhyay, A., Bhasin, S.: Side-channel assisted existential forgery attack on dilithium - a NIST PQC candidate. ePrint
31. Ravi, P., Yang, B., Bhasin, S., Zhang, F., Chattopadhyay, A.: Fiddling the twiddle constants - fault injection analysis of the number theoretic transform. CHES (2023)
32. Rivest, R., Shamir, A., Adleman, L.: A method for obtaining digital signatures and public-key cryptosystems. Commun. ACM (1978)
33. Seifert, J.P.: On authenticated computing and rsa-based authentication. In: CCS (2005)
34. Shor, P.: Algorithms for quantum computation: discrete logarithms and factoring. In: FOCS (1994)
35. Soni, D., Basu, K., Nabeel, M., Aaraj, N., Manzano, M., Karri, R.: FALCON, pp. 31–41. Springer International Publishing, Cham (2021)
36. Timmers, N., Spruyt, A., Witteman, M.: Controlling pc on arm using fault injection. In: FDTC (2016)
37. Trouchkine, T., Bouffard, G., Clédière, J.: EM fault model characterization on socs: From different architectures to the same fault model. In: FDTC (2021)
38. Witteman, M.: Secure application programming in the presence of side channel attacks
39. Yuce, B., Schaumont, P., Witteman, M.: Fault attacks on secure embedded software: Threats, design and evaluation. CoRR (2020)
40. Zussa, L., Dutertre, J.M., Clédière, J., Robisson, B., Tria, A.: Investigation of timing constraints violation as a fault injection means. In: DCIS (2012)