Correction Fault Attacks on Randomized CRYSTALS-Dilithium

Elisabeth Krahmer[1,2], Peter Pessl[2], Georg Land[1] and Tim Güneysu[1,3]

[1] Ruhr-University Bochum, Bochum, Germany,
{elisabeth.krahmer,tim.gueneysu}@rub.de,mail@georg.land
[2] Infineon Technologies AG, Munich, Germany, peter.pessl@infineon.com
[3] DFKI GmbH, Bremen, Germany

Abstract. After NIST's selection of Dilithium as the primary future standard for quantum-secure digital signatures, increased efforts to understand its implementation security properties are required to enable widespread adoption on embedded devices. Concretely, there are still many open questions regarding the susceptibility of Dilithium to fault attacks. This is especially the case for Dilithium's randomized (or hedged) signing mode, which, likely due to devastating implementation attacks on the deterministic mode, was selected as the default by NIST.
This work takes steps towards closing this gap by presenting two new key-recovery fault attacks on randomized/hedged Dilithium. Both attacks are based on the idea of correcting faulty signatures after signing. A successful correction yields the value of a secret intermediate that carries information on the key. After gathering many faulty signatures and corresponding correction values, it is possible to solve for the signing key via either simple linear algebra or lattice-reduction techniques. Our first attack extends a previously published attack based on an instruction-skipping fault to the randomized setting. Our second attack injects faults in the matrix $\mathbf{A}$, which is part of the public key. As such, it is not sensitive to side-channel leakage and has, potentially for this reason, not seen prior analysis regarding faults.
We show that for Dilithium2, the attacks allow key recovery with as little as 1024 and 512 faulty signatures, respectively, with each signature generated by injecting a single targeted fault. We also demonstrate how our attacks can be adapted to circumvent several popular fault countermeasures with a moderate increase in the computational runtime and the number of required faulty signatures. These results are verified using both simulated faults and clock glitches on an ARM-based microcontroller.
The presented attacks demonstrate that also randomized Dilithium can be subject to diverse fault attacks, that certain countermeasures might be easily bypassed, and that potential fault targets reach beyond side-channel sensitive operations. Still, many further operations are likely also susceptible, implying the need for increased analysis efforts in the future.

Keywords: Fault Injection Attack · Dilithium · Post-Quantum Cryptography

1 Introduction

In 2022, NIST announced the first algorithms that will be standardized in the context of their post-quantum cryptography process [NIS22]. Out of the three selected signature algorithms, CRYSTALS-Dilithium [BDL+21] (renamed to ML-DSA in the recently published draft standard FIPS204 [NIS23]) was identified by NIST as the primary algorithm implemented for most use cases and is, compared to the other selected algorithms Falcon [FHK+20] and SPHINCS+ [ABB+22], arguably the most suitable for computing quantum-secure signatures on embedded devices.

Exactly such embedded devices are potential targets of physical attacks like side-channel analysis and fault attacks. As the real-world adoption of the NIST-selected algorithms might be imminent, gaining an understanding of the attack landscape and subsequently designing hardened implementations is a high-priority task.

The side-channel aspect of Dilithium has already seen extended analysis and the general attack landscape is somewhat understood (see, e.g., the 2023 survey by Ravi et al. [RCDB23] and the sensitivity analysis in [ABC+23]). Also, first works describing (higher-order) masked implementations are starting to appear [ABC+23, CGTZ23].

For fault attacks, however, the situation is different. In this regard, it is important to note that the Dilithium submission [BDL+21] designated two signing modes: *deterministic* and *randomized*. The deterministic signing mode prevents nonce reuse attacks due to poor randomness generation, but is known to be extremely vulnerable to differential fault attacks [BP18]. Arguably, deterministic signing also provides less favorable side-channel properties (due to the possibility of, e.g., averaging).

Likely due to these reasons, the so-called *hedged* mode was introduced and made the default in the first draft of the Dilitihium standard [NIS23]. Analogously to the previously mentioned randomized mode, the introduction of fresh randomness in each signing operation greatly increases robustness against physical attacks. Therefore, deterministic signing should not be used on platforms where physical attacks are of concern, according to NIST. This then puts fault analysis of the randomized/hedged mode into focus.[1]

There do exist some prior works demonstrating fault attacks on randomized Dilithium (or one of its progenitor schemes), see [RCDB23] for an overview. Ravi et al. [RJH+19], for instance, induce an instruction skip during an addition, which allows them to perform recovery after observing many faulty signatures. Whether their attack applies to the randomized mode depends on implementation details, such as the ordering of the additions. Later, Ravi et al. [RYB+23] demonstrated a fault attack that works by zeroing certain constants during loading. One of the attack variants applies to randomized Dilithium, but only if its implemented in a certain manner. As a final example, Islam et al. [IMS+22] describe a method utilizing bit flips in the secret key injected through Rowhammer. They perform key recovery by using the signature verification procedure as a correction oracle.

These works only target a selected few operations computed during signing, leaving a large space of yet unexplored sections open. Moreover, some of the attacks only apply to highly specific implementation variants of Dilithium. Finally, note that all mentioned methods target operations that more or less directly involve secret information, which makes them somewhat obvious targets for fault attacks. Also, the side-channel countermeasures needed to protect these operations might already make fault injections more difficult.

This leaves open the question of how the attacks fare under the consideration of standard (side-channel) countermeasures and if there are additional and potentially less obvious faulting targets for randomized Dilithium.

Our contribution. This work addresses the above questions by significantly extending the applicability of fault attacks on randomized Dilithium. In particular, we

- propose two new fault attacks extracting information of the secret by correcting faulty values with the help of the verify function.
- present a generalization of the skipping fault attack [RJH+19] capable of attacking randomized signing regardless of the addition order.
- describe the first key-recovery fault attack on Dilithium targeting a non-secret computation (the expansion of the public matrix $\hat{\mathbf{A}}$) and show how lattice-reduction methods can be used to reduce the number of required faults.

[1] In the context of this paper, the hedged and randomized mode behave identical, which is why we do not further distinguish between them in the following.

- show modifications of these attacks that can circumvent certain countermeasures, such as shuffling or sign-then-verify, under a slight to moderate increase in the number of required faulty signatures or computational runtime.
- verify the functionality of our attacks using simulated and real faults and show that as little as 512 faulty signatures can suffice for key recovery.
- provide the source code for the attacks at https://github.com/Chair-for-Security-Engineering/dilithium-faults

2 Preliminaries

This section presents the necessary background for understanding the new attacks. After recalling Dilithium, we discuss previously proposed attacks and countermeasures.

2.1 Dilithium

CRYSTALS-Dilithium [BDL+21] is a lattice-based digital signature algorithm and was selected as the primary future algorithm for quantum-secure signing by NIST [NIS22]. It will be standardized under the name ML-DSA, a first draft has recently been published [NIS23].

Dilithium operates on vectors of polynomials over the ring $\mathbb{Z}_q \equiv \{0, ..., q-1\}$, with $q = 8380417 = 2^{23} - 2^{13} + 1$ being a (fixed) 23 bit prime. Polynomials with coefficients in $\mathbb{Z}_q$ are denoted by lowercase letters: $z \in \mathbb{Z}_q[X] = \sum_i z[i]X^i$, with $\forall i : z[i] \in \mathbb{Z}_q$. We use $\mathcal{R}_q$ for the ring of polynomials in $\mathbb{Z}_q[X]$ reduced by $X^n + 1$, with $n = 256$ being fixed for all Dilithium parameter sets. Further, the subset S_η includes all coefficient-wise small polynomials: $S_\eta = \{z \in \mathcal{R}_q \,|\, \forall i = 0, ..., n-1 : \; -\eta \leq z[i] \leq \eta\}$. Concretely, Dilithium uses $\eta \in \{2, 3\}$ for its secret keys. Elements that are vectors (resp. matrices) of polynomials are written with lowercase (resp. uppercase) bold letters: $\mathbf{z} \in \mathcal{R}_q^k$, $\mathbf{A} \in \mathcal{R}_q^{k \times \ell}$.

Interpreting polynomials as (column) vectors of their coefficients allows to write polynomial multiplication as vector-matrix-multiplication: $ab \mod (X^n + 1) = Ab = Ba$ with the entries of A (resp. B) containing the coefficients of a (resp. b), rotated in a nega-cyclic manner. This specific rotation is used in a later section, and is referred to as $\mathsf{rot_{mult}} : \mathbb{Z}_q^n \times \{0, ..., n-1\} \to \mathbb{Z}_q^n$, $\mathsf{rot_{mult}}(a, t) = (a[t], a[t-1], ..., a[0], -a[n-1], -a[n-2], ..., -a[t+1])$. That is, $\mathsf{rot_{mult}}(a, t)$ returns the t-th row of the matrix A.

Dilithium prescribes the use of the *Number Theoretic Transform (NTT)* to implement polynomial multiplication efficiently. This operation is a generalization of the discrete Fourier transformation based upon a quotient ring instead of the field of complex numbers. Usage of the NTT allows to perform polynomial multiplications via point-wise multiplications of the transformed operands, thereby speeding up the multiplication operation significantly. We use $\hat{a} = \mathsf{NTT}(a)$ to denote the NTT domain representation of a polynomial a. For a vector of polynomials $\mathbf{v} \in \mathcal{R}_q^\ell$, the notation $\hat{\mathbf{v}} = \mathsf{NTT}(\mathbf{v})$ stands for the component-wise application of the NTT. Point-wise multiplication of transformed polynomials is written as $\hat{a} * \hat{b}$.

We now give a brief overview of the main algorithms but refer to the specification [BDL+21, NIS23] for further details on the parameter sets and auxiliary functions.

Key Generation. The algorithm starts by sampling a random matrix $\mathbf{A} \in R_q^{k \times \ell}$ and the main secret $(\mathbf{s_1}, \mathbf{s_2}) \in S_\eta^\ell \times S_\eta^k$. The main operation of keygen is to calculate the value $\mathbf{t} \in \mathcal{R}_q$ by multiplying $\mathbf{A} \cdot \mathbf{s_1}$ and then adding $\mathbf{s_2}$, making $\mathbf{t}$ a slightly disturbed version of $\mathbf{As_1}$. The output of keygen is the keypair (pk, sk).

Simply speaking, the public key pk consists of $\mathbf{A}$ and $\mathbf{t}$. However, Dilithium uses several techniques to reduce the size of the public key. The matrix $\mathbf{A}$ is generated deterministically from a seed ρ, so just publishing ρ is sufficient. The vector $\mathbf{t}$ is coefficient-wise split into

its high and low-order bits $\mathbf{t_1}$ and $\mathbf{t_0}$, respectively. Only the high bits $\mathbf{t_1}$ are published. The secret key sk includes $\rho, \mathbf{s_1}, \mathbf{s_2}$, and $\mathbf{t_0}$. Additionally, it contains two further values K and tr. The seed K is used as additional input for (deterministic or hedged) nonce derivation. The public-key hash tr is just included for efficiency reasons.

Signature Generation (Alg. 2.1). First, a nonce $\mathbf{y}$ is sampled and multiplied with the (public) matrix $\mathbf{A}$. The result $\mathbf{w} = \mathbf{Ay}$ is split into its high and low bits, and the high bits $\mathbf{w_1} = \mathsf{HighBits}_q(\mathbf{w})$ are hashed together with information on the message to form the challenge c. Then, the signer calculates the second part of the signature, $\mathbf{z} = \mathbf{y} + c\mathbf{s_1}$. After performing several checks on bounds of the variables and restarting signing if they are not met (Fiat-Shamir with aborts paradigm [Lyu09]), the hint $\mathbf{h}$ (the final signature component) is computed. The inclusion of the hint is required so that the verification procedure can compensate the omission of $\mathbf{t_0}$ in the public key. The hints describe if this omission causes an unwanted carry bit in a part of verification, allowing verification to correct the carry again.

In the context of this work, an important step of the sign algorithm is the generation of the seed ρ' that is later used to derive $\mathbf{y}$. In the recently introduced *hedged* mode [NIS23], ρ' is computed through hashing a concatenation of the secret seed K, the message representative μ, and the random string rnd. This construction provides robustness against nonce-reuse attacks while introducing fresh randomness in each signature generation and thus mitigates, e.g., differential fault attacks. For this reason, the hedged mode is now the default in FIPS204, whereas the optional deterministic mode should not be used on platforms where side-channel and fault attacks are of concern [NIS23].

Algorithm 2.1: Dilithium sign [BDL+21, NIS23]

Input: Secret key sk, message M

$\mathbf{A} \in R_q^{k \times \ell} := \mathsf{ExpandA}(\rho)$; // target Section 5

$\mu \in \{0,1\}^{512} := \mathsf{H}(tr||M)$;

$\kappa := 0, (\mathbf{z}, \mathbf{h}) := \perp$;

$rnd \leftarrow \{0,1\}^{256}$ (or $rnd := \{0\}^{256}$ for the optional deterministic variant);

$\rho' \in \{0,1\}^{512} := H(K||rnd||\mu)$;

while $(\mathbf{z}, \mathbf{h}) = \perp$ **do**

  $\mathbf{y} \in S^{\ell}_{\gamma_1} := \mathsf{ExpandMask}(\rho', \kappa)$;

  $\mathbf{w} := \mathbf{Ay}$;

  $\mathbf{w_1} := \mathsf{HighBits}_q(\mathbf{w}, 2\gamma_2)$;

  $\widetilde{c} \in \{0,1\}^{256} := \mathsf{H}(\mu||\mathbf{w_1})$;

  $c \in B_\tau := \mathsf{SampleInBall}(\widetilde{c})$;

  $\mathbf{z} := \mathbf{y} + c\mathbf{s_1}$; // target Section 4

  $\mathbf{r_0} := \mathsf{LowBits}_q(\mathbf{w} - c\mathbf{s_2}, 2\gamma_2)$;

  if $||\mathbf{z}||_\infty \geq \gamma_1 - \beta$ or $||\mathbf{r_0}||_\infty \geq \gamma_2 - \beta$ **then**

    $(\mathbf{z}, \mathbf{h}) := \perp$

  else

    $\mathbf{h} := \mathsf{MakeHint}_q(-c\mathbf{t_0}, \mathbf{w} - c\mathbf{s_2} + c\mathbf{t_0}, 2\gamma_2)$;

    if $||c\mathbf{t_0}||_\infty \geq \gamma_2$ or the # of 1's in $\mathbf{h}$ is $> \omega$ **then**

      $(\mathbf{z}, \mathbf{h}) := \perp$

  $\kappa := \kappa + \ell$;

return Signature $\sigma = (\widetilde{c}, \mathbf{z}, \mathbf{h})$

Signature Verification (Alg. 2.2). For verification, the verifier recovers the intermediate value $\mathbf{w_1}$ from the signing process. They achieve that by computing $\mathbf{Az} - c\mathbf{t_1} \cdot 2^d$, the hint $\mathbf{h}$ allows to compensate for the missing term $c\mathbf{t_0}$. They then compare the given challenge to the one derived by their calculation. If they are equal and all prescribed bounds are met, the signature is accepted as valid.

Algorithm 2.2: Dilithium verify [BDL+21, NIS23]

Input: Public key pk, message M, signature $\sigma = (\widetilde{c}, \mathbf{z}, \mathbf{h})$

$\mathbf{A} \in R_q^{k \times \ell} := \mathsf{ExpandA}(\rho)$;

$\mu \in \{0,1\}^{512} := \mathsf{H}(\mathsf{H}(\rho||\mathbf{t_1})||M)$;

$c := \mathsf{SampleInBall}(\widetilde{c})$;

$\mathbf{w_1'} := \mathsf{UseHint}_q(\mathbf{h}, \mathbf{Az} - c\mathbf{t_1} \cdot 2^d, 2\gamma_2)$;

if $\widetilde{c} = \mathsf{H}(\mu||\mathbf{w_1'})$ and $||\mathbf{z}||_\infty < \gamma_1 - \beta$ and # of 1's in $\mathbf{h}$ is $\leq \omega$ **then**

  accept

2.2 Implementation Attacks Targeting Dilithium

We now give an overview over previously proposed implementation attacks on Dilithium, we focus on signature generation.

Side-Channel Attacks. There exist several previous works presenting side-channel attacks on Dilithium. These range from DPA on point-wise multiplication with the key polynomials $(\mathbf{s_1}, \mathbf{s_2})$ [CKA+21] to attacks exploiting single-trace leakage of various operations [MUTS22, LZS+21, BVC+23]. For a more in-depth overview, we refer to [RCDB23].

The mentioned works only cover a small subset of likely susceptible operations, meaning that many more attack variations and improvements are to be expected. Still, the leakage sensitivity analysis in [ABC+23] shows which algorithm parts require protection, thereby giving a solid baseline for the design of countermeasures.

Fault Attacks. In 2018, Groot Bruinderink and Pessl [BP18] showed that the deterministic version of Dilithium is highly vulnerable to **differential fault attacks**. An adversary is assumed to be able to inject a single random fault during the signature generation to a message m. Then, they let the same message m be signed again, this time without any intervention. By injecting the fault after the deterministic computation of the nonce $\mathbf{y}$, a fault-induced nonce-reuse scenario is achieved, allowing trivial key recovery using a single faulty signature. The analysis of the attack shows that the single random fault can be injected in up to 65% of the overall signing time. This large timeframe, together with the relaxed fault requirements (single random fault) and the need to retrieve just a single faulty signature, make this a strong attack. However, it strictly applies only to deterministic signing, as it relies on the fact that the same $\mathbf{y}$ is generated when signing the same message twice.

[IMS+22] proposed a fault injection attack that is applicable to both the deterministic and randomized version. It uses a **signature correction approach** to recover the secret key part $\mathbf{s_1}$ bit by bit. For the attack, a single bit in $\mathbf{s_1}$ is flipped before the operation $\mathbf{z} = \mathbf{y} + c\mathbf{s_1}$. When a message is signed with the faulty secret key, the resulting signature does not verify, which gives room for the correction idea. As $\mathbf{s_1}$ is used exactly once in the sign function and the fault consists of a single-bit flip, both the fault position and the actual value of the faulted bit have a unique effect on the signature. Both are recovered by successively adding and subtracting rotations of the challenge c, multiplied by factors of $2^k, k = 0, ..., 31$, to different polynomials of the faulty $\mathbf{z'}$, testing each time for signature

validity. When the modified signature verifies, the rotation and vector indices reveal the position of the faulty coefficient in $\mathbf{s_1}$, while the found k corresponds to the fault's bit index. The sign of the modification indicates the original bit value. The adversary repeats this process until all bits in $\mathbf{s_1}$ have been flipped at least once, or until enough bits of $\mathbf{s_1}$ have been recovered so that finding the remaining ones can be efficiently done with other techniques. In experiments targeting PCs using Rowhammer as fault injection technique, the authors achieve a security level decrease from 2^{141} to 2^{89} against classical attackers and from 2^{128} to 2^{81} against quantum adversaries. These still high levels are due to the fact that they were not able to flip every bit in $\mathbf{s_1}$ using a Rowhammer attack. Still, the key-recovery technique is independent of the fault-injection technique.

Ravi et al. [RYB+23] proposed an attack that works by setting the constants used in the NTT (twiddle factors) to zero. They achieve that by manipulating the pointer to these constants such that read accesses return zero. Out of the two attack variants targeting Dilithium, only one is applicable to randomized signing. In addition, this attack variant requires that Dilithium is implemented in a specific manner. Concretely, the output $\mathbf{z}$ needs to be computed as $\mathbf{z} = \mathsf{NTT}^{-1}(\hat{\mathbf{y}} + \hat{c} * \hat{\mathbf{s}}_\mathbf{1})$, i.e., the addition must be computed in the NTT domain.

The Skipping Fault Attack. The first new attack presented in this work (Section 4) is an extension of the skipping fault attack by Ravi et al. [RJH+19], we therefore explain their approach in more detail.[2] Their fault attack targets line 11 of the Dilithium sign routine (Alg. 2.1), where $\mathbf{z}$ is set to $\mathbf{y} + c\mathbf{s_1}$. Concretely, they skip the addition of one single target coefficient of one target component through, e.g., injecting an instruction skip or disturbing the loading of an operand. After injecting this fault in $\ell \cdot n$ signing operations, the secret key part $\mathbf{s_1}$ can be recovered.

To indicate faulted values, we introduce additional notation: x' is the disturbed version of a value x. Let j be the index of the targeted vector component, and i be the index of the targeted coefficient. Faulting the signature generation in the described manner returns $\sigma' = (\widetilde{c}, \mathbf{z}', \mathbf{h})$ with $\mathbf{z}' = (z_0, ..., z'_j, ..., z_{\ell-1})$ and, depending on the implementation of the addition, $z'_j[i] = y_j[i]$ or $z'_j[i] = (c\mathbf{s_1})_j[i]$. The attack then proceeds according to this implementation choice.

- **Case I:** $(z')_j[i] = (c\mathbf{s_1})_j[i]$**.** The returned $(z')_j[i]$ is $(c\mathbf{s_1})_j[i] = \langle \mathsf{rot}_{\mathsf{mult}}(c, i), (s_1)_j \rangle$. The attacker chooses i, j and knows $(z')_j[i]$ and c through the faulty signature. Hence, they can generate a linear equation with n unknowns – the n coefficients of $(s_1)_j$. The attack generates n such equations by, e.g., faulting the signing of different messages or targeting different coefficients i. Note that this applies to both probabilistic and deterministic signing. Then, Gaussian elimination solves the resulting system for the unknown $(s_1)_j$. This process reveals the entire $\mathbf{s_1}$ when it is repeated for all ℓ vector components.

- **Case II:** $(z')_j[i] = y_j[i]$**.** In this scenario, the attack of [RJH+19] relies on properties of the deterministic signing mode. Namely, in addition to signing a message m with the faulted sign routine, they also let the device sign m correctly. Due to deterministic generation of $\mathbf{y}$, subtracting $\mathbf{z}'$ from $\mathbf{z}$ yields $(y_j[i] + c\mathbf{s_1}_j[i]) - y_j[i] = c\mathbf{s_1}_j[i]$. Hence, they can again set up a linear equation in the coefficients of $(s_1)_j$ and continue analogously to case I.

[2] The loop-abort attack presented in [EFGT18] can be seen as a variant of the skipping fault, which is why we do not further discuss it here. Additionally, loop-aborts often do not result in the desired result, which is the skipped coefficients being zero [RJH+19]. The *Skipping the Addition of the Randomness* attack presented in [BBK16] can also be seen as a variant of the skipping fault; it assumes that the addition of the entire $\mathbf{y}$ is skipped.

Signature Forgery with Partial Secret Knowledge. Many attacks, including those introduced by this work, recover the secret key part $\mathbf{s_1}$ when successful. Although this does not give the attacker knowledge of the whole secret signing key, it enables them to sign arbitrary messages, as shown in [BP18, RJH+19].

2.3 Countermeasures

We now give a brief overview of side-channel and fault countermeasures that have either already been used with or can be easily extended to Dilithium.

Shuffling randomizes the ordering of independent computations (cf. [VMKS12]). Due to the high level of independent operations–many computations are performed on individual coefficients–shuffling can be used extensively and with low computational overhead in a large range of lattice-based schemes. Shuffling is particularly effective against algebraic side-channel and fault attacks, as it can obscure which exact coefficient has been measured/faulted.

Masking is arguably the most prominent side-channel countermeasure; the first fully high-order masked Dilithium implementations have already been presented ([ABC+23, CGTZ23]). While primarily aimed at obstructing passive attacks, masking can, in certain scenarios, be a hindrance to fault attacks as well. Some attacks require injecting the same fault at the same position in the processing of each share. Performing the previously described skipping fault, for instance, requires skipping over the addition of all individual shares of a specific value. This can severely complicate fault injection, especially if masking is combined with share-individual shuffling.

There has been relatively little analysis of specifically fault-related countermeasures for lattice-based cryptography. Some generic techniques exist, such as **double computation** and **sign-then-verify**, and ad-hoc techniques for protecting against specific attacks have been proposed. Double computation, i.e., computing the signature twice and checking the results for equality, can exploit Dilithium's rejection structure: only the last iteration needs to be recomputed, pushing the (average) computational overhead below the typical factor of 2. Sign-then-verify benefits from the fact that verification is less costly than signing [KPR+], especially for secured implementations, but requires storing the (large) public key alongside the secret key.

As for a countermeasure more tailored towards the scheme while still providing somewhat generic protection, [HP23] proposed a method utilizing the **Chinese remainder theorem**. After lifting all input coefficients to an extension ring (using predetermined constants), one can perform all (modular) arithmetic operations in said extension ring and finally test if the result reduced by the extension modulus matches a precomputed value. The evaluation of the authors shows a computational overhead of 70% when applying the method to the Kyber NTT.

3 Attack Concept

As shown in the previous section, there are still significant gaps in the landscape of implementation attacks on Dilithium. First, some attacks require Dilithium to be implemented in a certain way and might be defeated with relatively simple countermeasures, such as shuffling. And second, most previously proposed fault attacks target the nonce $\mathbf{y}$ or the computation of $\mathbf{z} = \mathbf{y} + c\mathbf{s_1}$ in one way or the other, leaving large chunks of the algorithm still to be analyzed.

We aim to decrease this gap by presenting two new attacks. In this section, we give a first high-level overview of their workings.

Attacker Model. We assume an adversary has physical access to a device running the Dilithium signing routine with an unknown secret but known public key. The adversary can trigger signing and gather the (not necessarily controllable) signed message as well as the resulting signature. To achieve the goal of secret key recovery, the adversary can inject a single targeted fault in each call to the signing routine. We specify more concrete details on the required faults in the following sections.

For the theoretic part of this work, we assume that fault injection always succeeds, but we discuss methods that could be used to at least detect failed injections in Section 6. Finally, for our analysis, we assume that the desired fault is always injected in the final iteration of Dilithium's rejection loop, if the targeted operation is performed inside this loop. An equivalent assumption is that the adversary can inject the same fault in every loop. While this assumption is not realistic (the number of required iterations cannot be predicted when using randomized signing), and the equivalent one contradicts a previous statement (single fault per signing call), they make the analysis much more straight forward. In addition, one can use simple arguments to extrapolate the results to a more general scenario. For instance, one can always fault the first iteration and only use the result if signing succeeds in this iteration (detectable through runtime). The number of expected faults then needs to be divided by the probability that the first iteration succeeds, which is 0.23, 0.19, and 0.26 for Dilithium2, 3, and 5, respectively. Note that similar arguments were given for previous fault attacks on Dilithium [EAB+23].

Attack Intuition. The differential fault attack [BP18] as well as main parts of the skipping fault [RJH+19] compute differences between a faulty and a correctly computed signature, which in these cases carries easily exploitable information on the secret key. However, these attacks rely on $\mathbf{y}$ being identical in both calls, which is only easy to achieve when using deterministic signing (by signing the same message twice).

The approach presented in this paper also works with differences between faulty and valid signatures. However, the two signatures are not generated by running the signing routine twice; it is called only once, with the fault injected. Due to the injected fault, the result of this call does not validate. The idea is to now modify – *correct* – chosen intermediates until the verification succeeds. Concretely, this correction is done by enumerating the potential values of the intermediate, computing its effect on the faulty signature, and then feeding the modified signature to the verification algorithm. We then use the required modification to gain information on the secret. A challenge of this approach is to find a faulting target that, under *feasible* faulting models, allows *efficient correction* and provides *exploitable information* on the secret key. Such a correction approach was previously used in [IMS+22], however, their target is the secret key part $\mathbf{s_1}$ itself (which is known to require well-protected storage). In this work, we show that this approach is much more versatile and can be applied to two further fault scenarios: the skipping fault and a fault in the public matrix $\mathbf{A}$.

Recall that the attack of [RJH+19] does not succeed if randomized signing is used and the fault has the effect that $(z')_j[i] = y_j[i]$ instead of $(z')_j[i] = y_j[i] + (c\mathbf{s_1})_j[i]$ (cf. case II in Section 2.2, also mentioned in [BBK16]). This scenario can now be attacked by exploiting the correction approach with our **first attack**, which is illustrated in Fig. 1. After obtaining a faulty signature (c, z', h), the attack works by exhaustively enumerating the potential values of $(c\mathbf{s_1})_j[i]$ for fixed i, j, adding it to $z'_j[i]$, and passing the result to verification. Only for the correct value of $(c\mathbf{s_1})_j[i]$ verification will succeed. The recovered value features a linear dependency in $\mathbf{s_1}$, thereby allowing trivial key recovery after gathering enough faulty signatures and corresponding correction values. We give more in-depth explanations in Section 4.

The **second attack**, illustrated in Fig. 2, targets the expansion of the matrix $\mathbf{A}$ from the seed ρ (line 1 of Alg. 2.1). Note that ρ (and thus $\mathbf{A}$) is public. Hence, the expansion

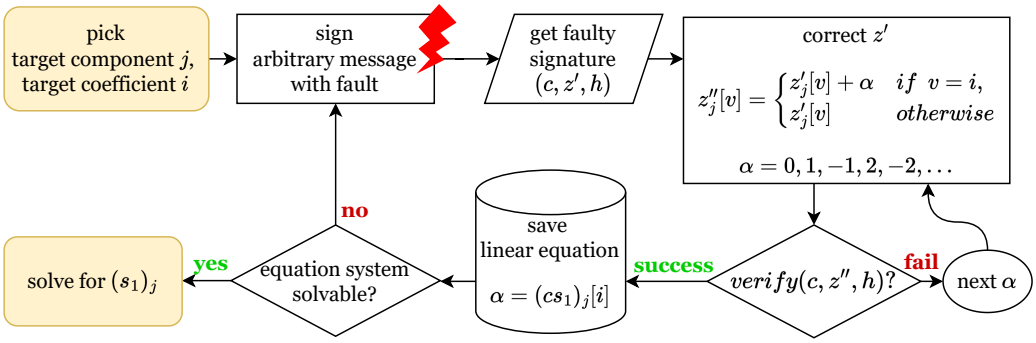

Figure 1: Illustration of our first proposed attack for the recovery of one $\mathbf{s_1}$ component.

can be computed without side-channel countermeasures in place, whereas the need for fault countermeasures was not analyzed previously. When using the modified $\mathbf{A}'$ during the signing process, the resulting signature will most likely not verify, which gives room to the correction approach. In contrast to the skipping fault scenario, we do not correct the signature (c', z', h') itself for this attack. Instead, we look at the essential equation of the Dilithium verification routine. More specifically, we aim at finding a value $\star$ such that $\mathsf{HighBits}_q(\mathbf{Az}' - c'\mathbf{t} + \star) = \mathbf{w}'_1$ (cf. Alg. 2.2), which corresponds to a verification success. As it turns out, the value of $\star$ can be expressed as $(\mathbf{A} - \mathbf{A}')\mathbf{y} = \Delta\mathbf{Ay}$. By inducing a $\Delta\mathbf{A}$ such that the values of $\star$ are practically enumerable, one can find the correct $\star$ and thereby retrieve information on $\mathbf{y}$ (and thus the secret key part $\mathbf{s_1}$). This attack approach is sketched in Figure 2, we discuss further attack details in Section 5.

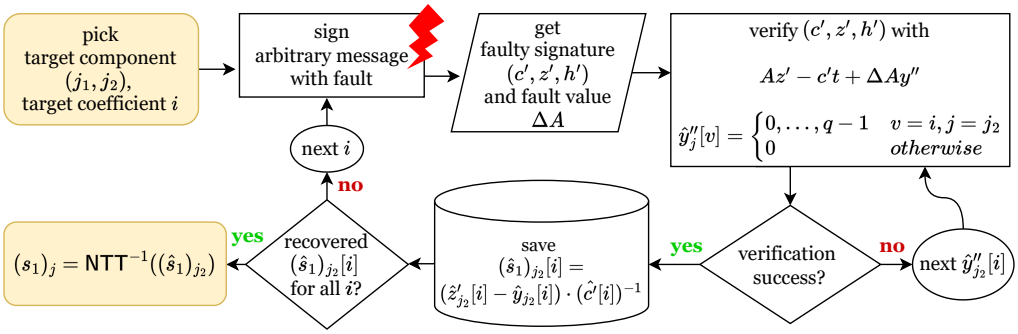

Figure 2: Illustration of our second proposed attack for the recovery of one $\mathbf{s_1}$ component.

4 Skipping Fault Correction Attack

Our first attack is an extension of [RJH+19] to successfully attack Dilithium independent of the used signing mode, deterministic or randomized.

4.1 Fault Model and Exploitation

For the attack to work, the attacker must be able to disturb the addition $\mathbf{z} = \mathbf{y} + c\mathbf{s_1}$ in exactly one targeted coefficient, such that the returned value equals one of the summands. This can be achieved through, e.g., skipping the addition instruction or by setting one summand to zero (or any other known constant value) by disturbing the load operation.

To adapt the concept by [RJH+19] to the scenario where the faulty signature returns $(z')_j[i] = y_j[i]$ (cf. case II in Section 2.2), we apply the correction idea. In general, the faulty signature is invalid and will not pass the verification. We know, however, that it will be valid if we correct the single coefficient that was targeted by the fault, $(z')_j[i] = y_j[i]$, to $(z')_j[i] = y_j[i] + (c\mathbf{s_1})_j[i]$. This correction is done by an adapted exhaustive search. The probability distribution for the values of the coefficients in $(c\mathbf{s_1})_j$ can be approximated by a (discretized) normal distribution with mean 0 and small variance, as shown in Fig. 3. Hence, the entries of $c\mathbf{s_1}$ are likely to be close to zero.

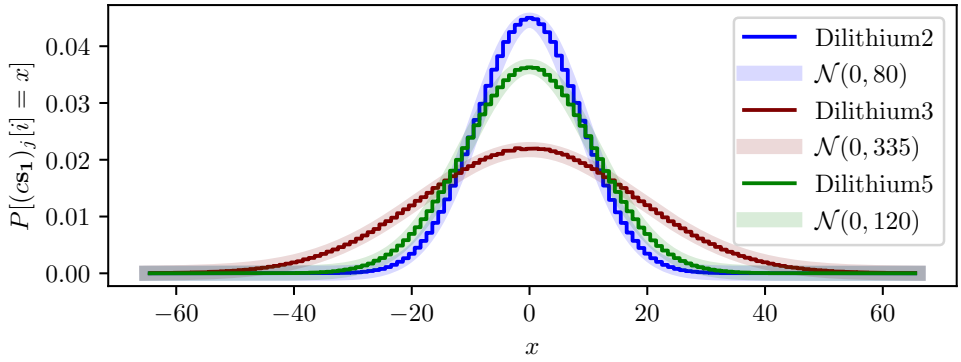

Figure 3: Probability distribution of the coefficients of $c\mathbf{s_1}$, simulated with 2^{25} samples. For comparison, the probability density functions of $\mathcal{N}(0, 80), \mathcal{N}(0, 335), \mathcal{N}(0, 120)$ are plotted alongside.

For the attack, the single coefficient $(z')_j[i]$ in the faulty signature σ' is modified by some $\alpha : (z'')_j[i] = (z')_j[i] + \alpha$, beginning with small absolute valued $\alpha = 1, -1, 2, -2, 3, ...$ After each modification, we run verify $(\widetilde{c}, \mathbf{z''}, \mathbf{h})$, with $\mathbf{h}, \widetilde{c}$ coming from the faulty signature. If the adjusted signature passes the verification, we assume that the tested α is indeed $(c\mathbf{s_1})_j[i]$. This directly allows us to set up a linear equation in the coefficients of $(s_1)_j$. Using this method, one can proceed as in [RJH+19], repeating the process to generate a solvable equation system. For the flexibility of the attack, we introduce an additional parameter $0 \leq b \leq \beta$, the *correction bound*, to cap the number of tested values α. A pseudo-code of the complete attack can be seen in Alg. 4.1.

4.2 Analysis

In the following, we analyze the attack with regards to complexity and fault injection aspects.

Complexity. The complexity of the attack is dominated by the number of required faults. For every component of $\mathbf{s_1}$, the attack needs to generate n linearly independent equations. In all our tests, we never encountered the case of generating linearly dependent equations.[3] Thus it appears reasonable to approximate the total number of equations to be generated and, subsequently, the number of required faults for the key recovery, with $\ell \cdot n = \{1024, 1280, 1792\}$. Recall, however, that depending on the setup, these numbers might need to be multiplied with the inverse of the probability that the first iteration yields a signature (cf. Section 3). This then yields, on average, 4500, 6600, and 7000 fault injections, respectively. In the (apparently very rare) case of encountering linear

[3] Using the formula of [Wat87] with the Dilithium parameters, we get that the probability of a random $n \times n$ matrices in $\mathbb{Z}_q$ being invertible is $\approx 1 - 1/q$. The specific distribution of c might slightly alter this result, however.

Algorithm 4.1: Skipping fault correction attack

Input: Public key pk, sign-with-fault-oracle $\mathcal{O}_{skip}$

set M to an arbitrary message
for $j = 0$ to $\ell - 1$ **do**
  $S := \{\}$;
  $target := (j, 0)$;
  while S not solvable **do**
    $(\widetilde{c}, \mathbf{z}', \mathbf{h}) := \mathcal{O}_{skip}(M, target)$ $\mathbf{z_{temp}} := \mathbf{z}'$;
    for $\alpha = 0, 1, -1, 2, -2, ...$ to $-b$ **do**
      $(z_{temp})_j[0] := (z')_j[0] + \alpha$;
      if verify $((\widetilde{c}, \mathbf{z_{temp}}, \mathbf{h})) = accept$ **then**
        $eq\_sol := \alpha$;
        $c :=$SampleInBall$(\widetilde{c})$;
        $eq :=$ rot$_{\mathsf{mult}}(c, 0)$;
        S.append$((eq, eq_sol))$;
        break;
  $(s_1)_j :=$ Solve(S)
return Secret key part $\mathbf{s_1}$

dependencies, one can generate a small number of additional equations or guess a small number of key coefficients. The exhaustive search space for the correction is bounded by $2\beta < 400$ and has therefore no significant impact on the complexity of the attack.

Fault Injection Aspects. The practicability of the skipping fault injection has been experimentally validated by [RJH+19]. Nonetheless, the skipping fault targets an operation that works on side-channel sensitive values. Thus, it is to be expected that the fault target is protected by countermeasures in real-world implementations, which can increase the difficulty of the fault injection.

For examining the influence of fault injection failures, we selected some scenarios for illustration. A failed injection could skip the addition in a coefficient that is different from the target (while affecting the signature). The failure could hit the target but disturb it in a way other than just skipping the addition, or it may affect a completely different operation. These scenarios are, in most cases, easily detectable since the exhaustive search will (with overwhelming probability) never be successful. Additionally, they highlight the main use of the correction bound b. If the fault injection was successful, the value to be found by the exhaustive search is likely to be close to zero. The longer the search takes, the higher the probability of a failed fault injection. Aborting the search earlier by lowering b lets the adversary balance the attack in that regard.

On the other hand, the scenario in which the fault injection misses completely is indistinguishable from a successful but ineffective fault, where the skipped addition was simply an "add zero" operation. To avoid those situations, the attack can be modified to exclude those ineffective faults.

4.3 Countermeasures

This section investigates the effects that selected protection methods have on the skipping fault correction attack and whether they can be circumvented with reasonable expenditure.

Shuffling. By randomly rearranging the order of the additions, which can be done at a comparatively low computational overhead, one can aim to hide which coefficient was

processed (and thus faulted) at any time. Still, our attack can be modified to circumvent the countermeasure at the cost of a moderately increased complexity and a slight increase in the number of faults by simply extending the exhaustive search. We correct every possible coefficient, the first test that verifies reveals the true skipped coefficient and the true missing value. Note that ineffective faults, i.e., skipped additions of zero, give us no possibility of recovering the actual fault position and therefore need to be discarded. The complete, shuffling-adapted attack (assuming the finest shuffling granularity) is depicted in Alg. 4.2, for simplicity with $b = \beta$. If a coarser shuffling is chosen and known to the attacker, the algorithm can be adapted accordingly.

Extending the brute-force search to every possible coefficient leads to an extra runtime factor of $\ell \cdot n$. Additionally, the number of required faults is increased by two effects. First, ineffective faults must be discarded. With our approximation of the coefficient distribution in $c\mathbf{s_1}$ (cf. Fig. 3), on average about 4.5% (resp. 2.2% for Dilithium3, 3.6% for Dilithium5) of all fault injections result in ineffective faults and cannot be used for the shuffling-adapted attack. Second, when attacking a shuffled addition, the adversary cannot choose the targeted component. For the attack, however, they need to collect n equations for each of the ℓ secret components of $\mathbf{s_1}$ to be able to solve the resulting equation systems. When they cannot control the target, they might need to retrieve more than the $(\ell \cdot n)$ faulty signatures to end up with ℓ solvable systems. Our tests show that both effects combined lead to fault overhead of around 10% to 14% (compared to the minimal number of required faults $\ell \cdot n$).

Algorithm 4.2: Skipping fault correction attack, circumventing shuffling

Input: Public key pk, sign-with-fault-oracle $\mathcal{O}_{skipShuff}$
1 **for** $j = 0$ to $\ell - 1$ **do**
2 $S_j = \{\}$;
3 set M to an arbitrary message;
4 **while** $\exists j$ s.t. S_j not solvable **do**
5 $(\widetilde{c}, \mathbf{z'}, \mathbf{h}) := \mathcal{O}_{skipShuff}(M, target = (0, 0))$;
6 **if** $\mathsf{verify}((\widetilde{c}, \mathbf{z'}, \mathbf{h})) = accept$ **then**
7 **continue**;
8 **for** $\alpha = 1, -1, 2, -2, ...$ to $\beta, -\beta$ **do**
9 **for** $j = 0, ..., \ell - 1$ **do**
10 **for** $i = 0, ..., n - 1$ **do**
11 $\mathbf{z_{temp}} := \mathbf{z'}$;
12 $(z_{temp})_j[i] := (z')_j[i] + \alpha$;
13 **if** $\mathsf{verify}\ ((\widetilde{c}, \mathbf{z_{temp}}, \mathbf{h})) = accept$ **then**
14 $eq_{sol} := \alpha$;
15 $c := \mathsf{SampleInBall}(\widetilde{c})$;
16 $eq := \mathsf{rot}_{\mathsf{mult}}(c, i)$;
17 $S.\mathsf{append}((eq, eq_{sol}))$;
18 **goto** next while iteration (Line 4)
19 **for** $j = 0$ to $\ell - 1$ **do**
20 $(s_1)_j = \mathsf{Solve}(S_j)$
21 **return** $\mathbf{s_1}$

Masking. As mentioned in Section 2.3, the costly masking countermeasure is primarily aimed at mitigating side-channel attacks, but can still increase the skipping fault correction

attack's complexity. To make the skipping fault correction attack work on a masked implementation, the attacker needs to succeed in faulting the addition in all shares. This can be a non-trivial challenge, especially when combining masking with (share-wise) shuffling. Note that the attacker is likely to be able to detect whether or not they have succeeded in faulting the same coefficient in all shares. Any partial addition skip leads to an effectively randomized $\mathbf{z}$, which has a high chance of being rejected during the signature generation and hence not being propagated to the output.

Double Computation and Sign-Then-Verify. While double computation detects most single-fault attacks, sign-then-verify inherently prevents all correction attacks. However, there is one way to circumvent both measures even in the single-fault model. The attack can be turned into an ineffective attack by setting the correction bound $b = 0$, only using faulted signatures if the fault had no consequence (the skipped addition was an "add zero" operation). This effectively voids the correction idea – there is no faulty value to be corrected. The modification of the attack from Alg. 4.3 is simple. Instead of ending up with a solvable linear equation system, the attack generates a system where the solution vector is 0, requiring us to calculate the kernel of the system matrix. The correct element of the kernel space is then found by trying all possible factors, first filtering out all kernel elements in S_η and then testing the remaining candidates to find $\mathbf{s_1}$. One way to do so is to sign a random message and see if the result verifies using the original public key.

The main complexity increase of this attack modification comes from the number of needed faults. The probability for the skipping fault to result in an ineffective fault is the probability of $(c\mathbf{s_1})_j[i]$ being zero. According to the simulation in Fig. 3, we can assume an upper bound for this value of 5% for Dilithium2, which leads to an expectation value of more than 20 fault injections per ineffective fault. For the parameter sets 3 and 5 this factor is even larger.

It is worth noting that the modifications of the skipping fault correction attack to counter shuffling and double computation/sign-then-verify are mutually exclusive. Adapting the attack to circumvent both countermeasures at the same time is not possible in the described way. Also, fault injection failures cannot be detected within the ineffective attack.

Fault Protection with Chinese Remainder Theorem [HP23]. When skipping the target addition in one coefficient in an implementation that is protected by lifting the calculations to $\mathcal{R}_{\hat{q}} \cong \mathcal{R}_q \times \mathcal{R}_{q'}$, it not only affects the addition result modulo q but also the outcome modulo q', making it differ from the pre-computed check values. The integrity checks of the countermeasure can therefore detect the skipping fault with high probability and discard any faulted computation, effectively preventing the attack. [HP23] report a computational overhead of 70%, although their analysis was focused solely on the Kyber NTT and thus might not match results for Dilithium.

NTT Addition. *NTT addition* was proposed by [RJH+19] to counteract the skipping fault on Dilithium specifically. Instead of calculating the addition $\mathbf{z} = \mathbf{y} + c\mathbf{s_1}$ directly, they suggest to perform the addition in the NTT domain, i.e., as $\mathbf{z} = \mathsf{NTT}^{-1}(\hat{\mathbf{y}} + \hat{c} * \hat{\mathbf{s}}_\mathbf{1})$. This change causes the faulted signature to be discarded with very high probability, preventing the fault from being propagated to the returned signature. Note, however, that this implementation change can enable other fault attacks [RYB+23].

5 Correction Attack with a Fault in A

In the previous section, the applicability of the correction approach was demonstrated using a fault equivalent to an instruction skip at one specific point during the signature calculation. The targeted operation, $\mathbf{z} = \mathbf{y} + c\mathbf{s_1}$, can be perceived as highly vulnerable to

Algorithm 4.3: Skipping fault correction attack, with ineffective faults

Input: Public key pk, sign-with-fault-oracle $\mathcal{O}_{skip}$

set M to an arbitrary message;
for $j = 0$ to $\ell - 1$ **do**
  $S := \{\}$;
  $\text{factors}_j := \{\}$;
  target $:= (j, 0)$;
  while S not solvable **do**
    $(\widetilde{c}, \mathbf{z}', \mathbf{h}) := \mathcal{O}_{skip}(M, target)$;
    if $\mathsf{verify}((\widetilde{c}, \mathbf{z}', \mathbf{h})) = reject$ **then**
      continue;
    $c := \mathsf{SampleInBall}(\widetilde{c})$;
    $eq := \mathsf{rot}_{\mathsf{mult}}(c, 0)$;
    $S.\mathsf{append}(eq)$;
  $(s_1)_j := s \in \ker(S)$;
  for $f = 1$ to $\frac{q-1}{2}$ **do**
    if $f \cdot (s_1)_j \in S_\eta$ **then**
      $factors_j.\mathsf{append}(f, -f)$
for $f = (f_0, ...f_{\ell-1}) \in \bigotimes_{j=0}^{\ell-1} \text{factors}_j$ **do**
  $\sigma := forge\_sig(f \cdot \mathbf{s_1}, pk, M)$;
  if verify $(pk, M, \sigma) = accept$ **then**
    return secret key part $f \cdot \mathbf{s_1}$

side-channel attacks as it directly involves the secret key $\mathbf{s}_1$. As such, it is not far-fetched to assume that it is secured by countermeasures against side-channel attacks. Hence, looking at the correction approach in the context of faults in potentially less protected operations seems enticing.

Fault Model. For our second attack, we require a fault injected in the public key part $\mathbf{A}$. The function ExpandA samples the matrix $\mathbf{A}$ from the seed ρ, directly returning the NTT domain representation. We assume that we are able to fault this process, resulting in a slightly disturbed $\hat{\mathbf{A}}' = \hat{\mathbf{A}} + \boldsymbol{\Delta}\hat{\mathbf{A}}$. More specifically, we only inject one single fault, which is described by $\boldsymbol{\Delta}\hat{\mathbf{A}}$ being zero everywhere except for exactly one coefficient of one entry of the matrix: $(\Delta\hat{A})_{j_1,j_2}[i] \neq 0$. Note that while the attacker does not need the ability to choose a specific fault value, he needs to know the injected difference $(\Delta\hat{A})_{j_1,j_2}[i]$.

This fault can be realized by, e.g., flipping/setting/resetting individual bits or setting the entire target coefficient to zero (or any other known value). While individual bit flips likely require precise fault injection, injecting a known value can be achieved through multiple paths, e.g., disturbing a load/store operation. These faults can be injected either in ExpandA, e.g., in the last round of SHAKE (used for extending the seed ρ) or in the rejection-sampling procedure of ExpandA, in the multiplication $\mathbf{Ay}$, or while $\mathbf{A}$ is kept in memory.

Note that if the attacked implementation generates $\mathbf{A}$ once at the beginning of signing and then keeps it in memory, i.e., it doesn't re-generate it in every iteration of the loop, then a single fault in ExpandA affects all iterations. Thus, in this case there is no need to filter for signatures that are returned after the first iteration (unlike in the skipping fault).

5.1 Fault Exploitation

To explain the proposed attack, we recall the details of Dilithium's signature verification (Alg. 2.2). The main idea of the verify function is to rebuild the intermediate value $\mathbf{w_1}$ that is created during the signing process. With $\mathbf{w_1}$, the verifier calculates the challenge $\widetilde{c}$ and checks for equality to the given signature.

For simplicity, we first allow the attacker to access $\mathbf{t}$ (and not just the high bits $\mathbf{t_1}$). Given the faulty signature σ' and the public key, they compute $\mathbf{A}, \mu$, and c' according to lines 1-3 of the verification procedure. Note that they reconstruct the correct matrix $\mathbf{A}$ during these steps, since they use ρ from the public key, but they can only recover the faulty c', as they only know the faulty $\widetilde{c}'$. Consequently, for the next step of verify, the following calculation is carried out: $\mathbf{Az}' - c'\mathbf{t} = \mathbf{A}(\mathbf{y} + c'\mathbf{s_1}) - c'(\mathbf{As_1} + \mathbf{s_2}) = \mathbf{Ay} - c'\mathbf{s_2}$.

To satisfy the main acceptance criterion in verify ($\widetilde{c} = \mathsf{H}(\mu||\mathbf{w_1'})$), we need to find the value $\mathbf{w}'$ such that $\mathsf{HighBits}_q(\mathbf{w}')$ equals the $\mathbf{w_1}'$ calculated during the signature generation. Exploiting the linearity of the NTT, this $\mathbf{w_1}'$ can be written as

$$\begin{aligned}
\mathbf{w_1}' &= \mathsf{HighBits}_q\left(\mathbf{w}', 2\gamma_2\right) \overset{\dagger}{=} \mathsf{HighBits}_q\left(\mathbf{w}' - c'\mathbf{s_2}, 2\gamma_2\right) = \mathsf{HighBits}_q\left(\mathbf{A}'\mathbf{y} - c'\mathbf{s_2}\right) \\
&= \mathsf{HighBits}_q\left(\mathsf{NTT}^{-1}\left((\hat{\mathbf{A}} + \boldsymbol{\Delta}\hat{\mathbf{A}}) * \mathsf{NTT}(\mathbf{y})\right) - c'\mathbf{s_2}\right) \\
&= \mathsf{HighBits}_q\left(\mathsf{NTT}^{-1}\left(\hat{\mathbf{A}} * \hat{\mathbf{y}}\right) + \mathsf{NTT}^{-1}\left(\boldsymbol{\Delta}\hat{\mathbf{A}} * \hat{\mathbf{y}}\right) - c'\mathbf{s_2}\right) \\
&= \mathsf{HighBits}_q\left(\mathbf{Ay} + \mathsf{NTT}^{-1}\left(\boldsymbol{\Delta}\hat{\mathbf{A}} * \hat{\mathbf{y}}\right) - c'\mathbf{s_2}\right).
\end{aligned}$$

The equality † is assured by range checks during the signing routine.

Note that $(\mathbf{Ay} - c'\mathbf{s_2})$ is exactly the expression computed earlier as part of the verification, i.e., $(\mathbf{Az}' - c'\mathbf{t})$. Thus, we require

$$\mathsf{HighBits}_q\left(\underbrace{\mathbf{Az}' - c'\mathbf{t}}_{\text{calculation by the verifier}} + \underbrace{\mathsf{NTT}^{-1}\left(\boldsymbol{\Delta}\hat{\mathbf{A}} * \hat{\mathbf{y}}\right)}_{\star}\right) = \mathbf{w_1}'. \tag{1}$$

The core idea of the attack is to find $\star$ by modifying – *correcting* – the value $\mathbf{Az}' - c'\mathbf{t}$ until the main verification condition is satisfied, i.e.,

$$\mathsf{H}(\mu||\mathsf{HighBits}_q(\mathbf{Az}' - c'\mathbf{t} \ + \mathsf{NTT}^{-1}(\boldsymbol{\Delta}\hat{\mathbf{A}} * \hat{\mathbf{y}})) = \widetilde{c}'.$$

As the only unknown in $\star$ is $\hat{\mathbf{y}}$, finding $\star$ is equivalent to finding $\hat{\mathbf{y}}$. As before, the correction is done via an exhaustive search. Since the fault is chosen such that $\boldsymbol{\Delta}\hat{\mathbf{A}}$ has exactly one non-zero entry, the correction value $\star$ can be found by enumerating the possible values of one single coefficient in $\hat{\mathbf{y}}$. Let (j_1, j_2) be the indices of the position in the matrix and i the coefficient index of the fault injected in $\hat{\mathbf{A}}$. Then

$$\star = \mathsf{NTT}^{-1}\left(\boldsymbol{\Delta}\hat{\mathbf{A}} * \hat{\mathbf{y}}\right) = \left(0, ..., 0, \ \mathsf{NTT}^{-1}\left((\Delta\hat{A})_{j_1,j_2} * \hat{y}_{j_2}\right), \ 0, ..., 0\right)$$

with, writing the polynomial as a vector containing its coefficients,

$$\mathsf{NTT}^{-1}\left((\Delta\hat{A})_{j_1,j_2} * \hat{y}_{j_2}\right) = \mathsf{NTT}^{-1}\left(\left(0, ..., 0, \ (\Delta\hat{A})_{j_1,j_2}[i] * \hat{y}_{j_2}[i], \ 0, ..., 0\right)\right).$$

The attack successively tests all values $0, 1, ..., q-1$ for $\hat{y}_{j_2}[i]$ in $\star$ and evaluates the left side of Eq. (1). If the result leads to a successful verification, we assume that it was equal to $\mathbf{w_1}'$ and that we have found the true coefficient $\hat{y}_{j_2}[i]$.

Finally, the knowledge of one coefficient $\hat{y}_{j_2}[i]$, together with the (faulty) signature and the linearity of the NTT reveals one coefficient of $(\hat{s}_1)_{j_2}$:

$$(\hat{s}_1)_{j_2}[i] = \left(\hat{z}'_{j_2}[i] - \hat{y}_{j_2}[i]\right) \cdot \left((\hat{c}')[i]\right)^{-1} \pmod q$$

Repeating this process for all entries in $\hat{\mathbf{s}}_\mathbf{1}$ and applying the inverse NTT reveals $\mathbf{s_1}$.

Influence of Key Compression. Up to now, we worked under the assumption that the attacker knows all of $\mathbf{t}$. However, the public key only contains the high bits $\mathbf{t_1}$. The missing information $\mathbf{t_0}$ is not regarded as sensitive information and can be reconstructed from a small number of signatures [NIS23]. Nonetheless, the attack still works if the adversary has access to only $\mathbf{t_1}$. The actual value for $\hat{y}_{j_2}[i]$ still always passes the verification test. Following the verify procedure, the attacker tries to correct the value $\mathbf{Az'} - c'\mathbf{t_1} \cdot 2^d$ (instead of $\mathbf{Az'} - c'\mathbf{t}$) by adding different $\mathbf{\Delta Ay}$. Let $\mathbf{y}^*$ be the nonce used during the signature generation. When the attacker tests the true value $\widehat{y^*}_{j_2}[i]$, they are effectively calculating

$$\mathbf{Az'} - c'\mathbf{t_1} \cdot 2^d + \mathbf{\Delta A y}^* = \mathbf{A}'\mathbf{y}^* + \mathbf{A}c'\mathbf{s_1} - c'\mathbf{t} + c'\mathbf{t_0} = \mathbf{A}'\mathbf{y}^* - c'\mathbf{s_2} + c'\mathbf{t_0},$$

which is exactly the value that was used during signing to create the hint (see line 16 in sign (Alg. 2.1)). Thus, $\mathsf{UseHint}_q\left(\mathbf{h}, \mathbf{Az'} - c'\mathbf{t_1} \cdot 2^d + \mathbf{\Delta A y}^*, 2\gamma_2\right) = \mathbf{w_1}$ and the verification succeeds. This proves that the attacker can always find the true value with the described brute-force search.

The full attack is summarized in the pseudo-code in Alg. 5.1 and 5.2. For simplicity, the row index j_1 of the targeted entry in $\hat{\mathbf{A}}$ is set to zero. Note that this choice is arbitrary, the value of j_1 does not matter for the attack and can even vary, it just needs to be known.

Algorithm 5.1: Modified verification verify'

Input: Public key pk, message M, signature $\sigma = (\widetilde{c}, \mathbf{z}, \mathbf{h})$, target (j_1, j_2, i), fault value δ, correction value α

1 $\mathbf{A} \in R_q^{k \times \ell} := \mathsf{ExpandA}(\rho)$;
2 $\mu \in \{0,1\}^{512} := H(H(\rho||\mathbf{t_1})||M)$;
3 $c := \mathsf{SampleInBall}(\widetilde{c})$;

4 $\mathbf{\Delta}\hat{\mathbf{A}} \in R_q^{k \times \ell} := \mathbf{0}$;
5 $(\Delta\hat{A})^i_{j_1,j_2} = \delta$;
6 $\hat{\mathbf{y}} \in R_q^k := \mathbf{0}$;
7 $\hat{y}^i_{j_2} = \alpha$;
8 $\mathbf{X} := \mathsf{NTT}^{-1}(\mathbf{\Delta}\hat{\mathbf{A}} * \hat{\mathbf{y}})$;

9 $\mathbf{w_1'} := \mathsf{UseHint}_q(\mathbf{h}, \mathbf{Az} - c\mathbf{t_1} \cdot 2^d + \mathbf{X}, 2\gamma_2)$;
10 **if** $\widetilde{c} = H(\mu||\mathbf{w_1'})$ and $||\mathbf{z}||_\infty < \gamma_1 - \beta$ and # of 1's in $\mathbf{h}$ is $\leq \omega$ **then**
11 **accept**

5.2 Reducing the Number of Needed Faults via Lattice Reduction

The method described above recovers exactly one coefficient of $\hat{\mathbf{s}}_\mathbf{1}$ per fault injection. This number can be significantly improved by exploiting the fact that all coefficients of $\mathbf{s}_1$ are small.

The NTT of a polynomial s can be described as a multiplication of its coefficient vector $\mathbf{s} \in \mathbb{Z}_q^n$ with the NTT-matrix $\mathbf{N} \in \mathbb{Z}_q^{n \times n}$. That is, $\hat{\mathbf{s}} = \mathbf{Ns}$ (and $\mathbf{s} = \mathbf{N}^{-1}\hat{\mathbf{s}}$). This is analogous to the discrete/fast Fourier transform and a multiplication with the DFT-matrix. When only recovering a subset of the coefficients of $\hat{\mathbf{s}}$, then $\mathbf{s}$ cannot be computed directly through the (inverse) NTT. Still, $\mathbf{s}$ can be found by exploiting its small coefficients.

W.l.o.g., we say that the first m coefficients in $\hat{\mathbf{s}}$ are known, we denote them as $\hat{\mathbf{s}}_k$. The $n - m$ unknown coefficients are grouped into $\hat{\mathbf{s}}_u$, giving us $\hat{\mathbf{s}} = \left(\hat{\mathbf{s}}_k^T || \hat{\mathbf{s}}_u^T\right)^T$. We can then

Algorithm 5.2: Correction attack with a fault in $\mathbf{A}$

Input: Public key pk, sign-with-fault-oracle $\mathcal{O}_{fA}$

set δ;
set M to an arbitrary message;
$\hat{\mathbf{s}}_\mathbf{1} = \mathbf{0}$;
for $j_2 = 0$ to $\ell - 1$ **do**
  for $i = 0$ to $n - 1$ **do**
    $target = (0, j_2, i)$;
    $(\tilde{c}', \mathbf{z}', \mathbf{h}') := \mathcal{O}_{fA}(M, target, \delta)$;
    for $\alpha = 0, 1, 2, ...$ to $q - 1$ **do**
      if verify'$(pk, M, (\tilde{c}', \mathbf{z}', \mathbf{h}'), target, \delta, \alpha) =$ accept **then**
        $(\hat{s}_1)_{j_2}[i] = (\hat{c}'_i)^{-1} \cdot ((\hat{z}'_{j_2})[i] - \alpha)$;
        break;
return $\mathsf{NTT}^{-1}(\hat{\mathbf{s}}_\mathbf{1})$

rewrite the NTT equation as

$$\mathbf{s} = \mathbf{N}^{-1}\hat{\mathbf{s}} = \begin{pmatrix} \mathbf{N}_k^{-1} & \mathbf{N}_u^{-1} \end{pmatrix} \begin{pmatrix} \hat{\mathbf{s}}_k \\ \hat{\mathbf{s}}_u \end{pmatrix}$$
$$\Leftrightarrow \quad -\mathbf{N}_u^{-1}\hat{\mathbf{s}}_u + \mathbf{s} = \mathbf{N}_k^{-1}\hat{\mathbf{s}}_k,$$

where $\mathbf{N}_k^{-1}$ and $\mathbf{N}_u^{-1}$ contain the columns of $\mathbf{N}^{-1}$ that are multiplied with $\hat{\mathbf{s}}_k$ and $\hat{\mathbf{s}}_u$, respectively. In this form, finding $\mathbf{s}$ can be seen as solving an LWE problem: the missing part of $\hat{\mathbf{s}}$ corresponds to the secret of the LWE instance, while $\mathbf{s}$ corresponds to the small added error. The NTT matrix part $\mathbf{N}_u^{-1}$ defines the lattice for the LWE problem.

If the known part of $\hat{\mathbf{s}}$ is sufficiently large, this LWE instance can be practically solved through lattice-reduction techniques. We have tested this approach, using the primal method as explained in [AGVW17], with an embedding factor of $t = 1$.

We note that a similar approach was previously presented in context of an attack on Kyber [HHP+21].

5.3 Analysis

Complexity. Recovering one coefficient of $\hat{\mathbf{s}}_\mathbf{1}$ requires injecting one fault in $\hat{\mathbf{A}}$ (at the corresponding row and coefficient index but in an arbitrary column) and then performing an exhaustive search over the values of one coefficient in $\hat{\mathbf{y}}$. The search space is of size q, which is much larger than the search space for the first correction attack ($2\beta \ll q$).

When not using lattice reduction, i.e., recovering each single coefficient of $\hat{\mathbf{s}}_1$ using a separate faulted signature, then a total of $\ell \cdot n = \{1024, 1280, 1792\}$ faults are required. Recall that in many implementations, $\mathbf{A}$ is generated before entering the rejection loop, meaning there is no need to specifically target the last iteration of said loop and to scale the numbers with the probability of the first iteration returning a signature.

With the approach described in Section 5.2, these numbers can be reduced drastically. Concretely, we were able to recover the key $\mathbf{s}$ with only half the coefficients of each polynomial in $\hat{\mathbf{s}}$ known, bringing down the total numbers to $\{512, 640, 892\}$. For computing the actual lattice reduction, we used the BKZ implementation of fplll [dt23] with a block size of 30. On a laptop equipped with an 13th Gen Intel i5 running at 1.6 GHz, the average runtime for recovering one key polynomial when half of its NTT coefficients are known was about 90 minutes. The expected runtime for recovering the entire $\mathbf{s}_1$ can be computed through multiplication with the parameter ℓ.

We ran 40 experiments using these parameters, all of them were successful. Note that we did not further optimize the parameters of this recovery step, thus even fewer faults will likely suffice.

Success Rate of the Key Recovery. The attack fails if a false value for $\hat{y}_j[i]$ leads to a verification. However, the properties of the NTT tell us that any wrong value for $\hat{y}_j[i]$ would lead to a deviation from the original $\mathbf{A}'\mathbf{y} - c'\mathbf{s_2} + c'\mathbf{t_0}$ in all coefficients. The lemma in Appendix A, on the other hand, shows that the $\mathsf{UseHint}_q$ function only allows very limited disturbances in its second input to not change its output. These two observations show that the probability of a false positive can reasonably be assumed to be negligible.

Practicability. A hurdle in practically realizing this attack is that the fault position corresponds to the recovered coefficient of $\mathbf{s_1}$ in a one-to-one relation, meaning that up to $\ell \cdot n$ different faulting positions/times are required. This stands in contrast to the attack from the previous section, where only ℓ different fault injection points are required. However, the expansion of $\mathbf{A}$ does not operate on any (side-channel) sensitive information and is, therefore, unlikely to be protected by additional countermeasures.

Fault-injection failures are potentially difficult to detect in this scenario, mainly because of the larger search space of the exhaustive search. In some fault injection failure cases, the brute-force search will test every possible value in $\mathbb{Z}_q$, ultimately failing every test. Additionally, our attack can only test for a correct value of $\star = \mathbf{\Delta A y}$. We use this property to find $\hat{\mathbf{y}}$, knowing the value of $\mathbf{\Delta}\hat{\mathbf{A}}$. However, any unknown deviation from the injected difference value or position significantly complicates the successful key recovery. It may be countered using redundancy similar to what we describe regarding shuffling circumvention. Regarding complete fault injection failures, i.e. the fault injection having no effect at all, the fault-$\mathbf{A}$ case does not differ from the skipping fault. It cannot be distinguished from the case in which the coefficient of $\hat{\mathbf{y}}$ that we try to learn is zero but may be circumvented by discarding all ineffective faults.

5.4 Countermeasures

When designing countermeasures for a fault attack on $\hat{\mathbf{A}}$, one has to consider that expanding $\hat{\mathbf{A}}$ makes up a significant part of the sign clock cycles. Moreover, for RAM-constrained devices, it might be required to component-wise re-compute $\hat{\mathbf{A}}$ for each iteration of the rejection loop. This induces a further significant slow-down of up to a factor of 6 [GKS21]. Thus, countermeasures should be inexpensive.

Shuffling. Each polynomial in $\hat{\mathbf{A}}$ is generated by using ρ and the matrix indices of the to respective polynomial as input for the extendable-output function (XOF) SHAKE-128. The output of the XOF is then subjected to rejection sampling to derive the individual coefficients (in increasing order of the indices). Said rejection sampling is an inherently serial process, which makes shuffling the order in which coefficients are generated difficult. Reordering the generation of the $(k \times \ell)$ polynomials within $\hat{\mathbf{A}}$, however, is trivial. Therefore, we limit our analysis to this second scenario.[4]

First, we note that determining the index of the currently-sampled entry of $\hat{\mathbf{A}}$ could be relatively simple when using side-channel leakage. Sampling a polynomial is a lengthy process (at least 5 calls to the Keccak-f permutation per polynomial), likely features minimal to no side-channel protection (it operates on public data), and its intermediates can be fully predicted (due to the public ρ). Thus, side-channel analysis could likely reveal which of the $(k \times \ell)$ entries is currently generated.

[4]The attack can also be adapted to the first scenario, albeit at higher cost. We did not analyze the exact overhead for this case.

Still, there are techniques to circumvent shuffling without resorting to side-channels, albeit at increased cost. First, consider the scenario that only the row indices of $\hat{\mathbf{A}}$ were shuffled. Analogously to the adaptation done for the skipping fault, one can simply test all k rows of $\mathbf{w_1}$, thereby increasing the computational runtime by this factor. Only applying a correction to the correct row will yield the correct $\tilde{c}'$.

If both row and column indices are unknown, additional steps are needed. Reconstructing $\mathbf{w}'_1$ (and finding the correct row j_1) is as costly as if only row indices are shuffled since finding the correct additive term $\Delta\hat{w}_1$ is sufficient. We assume that the adversary knows the potential $\Delta\hat{\mathbf{A}}_{j_1,j_2}[i]$ for all j_2, i.e., for the entire row; these values might depend on the actual fault index as is the case if the target coefficient i is faulted to zero. Then, we can compute $\hat{y}_{j_2}[i] = (\Delta\hat{w}_1) \cdot (\Delta\hat{A}_{j_1,j_2}[i])^{-1} \bmod q$, for all j_2. Only one of these ℓ entries in $\hat{\mathbf{y}}$ is correct. To determine which one is correct, we compute all corresponding $(\hat{s}_1)_{j_2}[i] = (\hat{z}'_{j_2}[i] - \hat{y}_{j_2}[i]) \cdot ((\hat{c}')[i])^{-1} \bmod q$. These candidates are saved, and the correction attack is repeated. Whenever we recover the same value for one coefficient in $\hat{\mathbf{s}}_\mathbf{1}$ twice, we can reasonably assume that we have found the correct value. Indeed, assuming an approximately uniform distribution of the $\hat{\mathbf{s}}_\mathbf{1}$ coefficients in $\mathbb{Z}_q$, the probability of recovering a coefficient incorrectly that way can be approximated by the birthday problem to around 2^{-15}.

To perform this attack, each column of $\hat{\mathbf{A}}$ must be hit at least twice. Using simulations, we found that this increases the number of required faulty signatures by a factor of 14, 19, and 29, for Dilithium 2, 3, and 5, respectively. The computational runtime is increased by an additional factor of k for finding the row index.

Double Computation and Sign-Then-Verify. Both double computation and sign-then-verify prevent the correction attack in the single fault model but might be ineffective when multiple faults can be injected precisely. The use of ineffective faults is significantly less applicable in this scenario, as such faults appear with probability of only $1/q \approx 2^{-23}$, which is likely too rare to allow a practical attack. However, the previously mentioned high runtime cost of generating $\hat{\mathbf{A}}$ might limit the usefulness of the double-computation approach. The sign-then-verify approach requires storing the public key alongside the private key, increasing storage requirements.

Fault Protection with Chinese Remainder Theorem. The CRT-based countermeasure [HP23] can only offer protection after lifting to an extension ring occurred. Thus, the sampling of $\mathbf{A}$ can, in most parts, not be protected using this countermeasure. Still, faults introduced on the stored (or re-loaded) $\mathbf{A}$ can very likely be detected.

6 Practical Evaluation

Both proposed attacks can be split into two parts: An online phase, in which faulty signatures are collected from the target, and an offline phase, in which these signatures are used to calculate the secret key part $\mathbf{s_1}$. We tested the key recovery phase for both attacks with simulated faults, before experimentally verifying the fault injection assumptions of the attack that faults the matrix $\mathbf{A}$.

6.1 Skipping Fault Correction Attack

To test the skipping fault correction attack (Section 4), we implemented the key recovery part of the attack based on the official reference implementation of Dilithium in C [ABD+]. The faults were simulated by programmatically injecting them into the `crypto_sign_signature` function of the reference. The tests were done on a laptop equipped with an 11th Gen Intel i5 2.6GHz CPU and the results are shown in Table 1.

Table 1: Recovery of s_1 using the skipping fault correction attack with simulated faults. Average was taken over 1000 (resp. *10) runs for each data point, recovery was successful in 100%.

Attack modification	Dilithium	∅ time	∅ # of faults	Ref.
plain attack	2	2.5 sec	1 024	Alg. 4.1
	3	7.9 sec	1 280	
	5	11.2 sec	1 792	
shuffling adapted*	2	28 min	1 162	Alg. 4.2
	3	2 h 22 min	1 414	
	5	5 h 39 min	2 041	
using ineffective faults	2	18 sec	22 952	Alg. 4.3
	3	62 sec	63 672	
	5	74 sec	49 514	

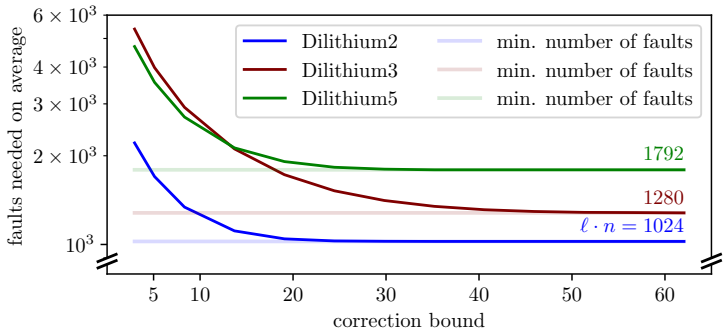

Figure 4: Average number of needed faults for the recovery of s_1 using the skipping fault correction attack, dependent on the correction bound. Average was taken over 200 runs for each data point, recovery was successful in 100%.

Note that, with this simulation approach, the instruction skip was triggered in every iteration of the rejection loop, and we therefore avoided the problem of not faulting the last iteration for testing the key recovery. As previously stated, the number of needed faults needs to multiplied by a factor of $\sim 4 - 5$ when going from simulated to actual faults.

In Section 4.3, we showed that the skipping fault correction attack can be adapted to an implementation that protects the target operation ($\mathbf{z} = \mathbf{y} + c\mathbf{s_1}$) with shuffling (Alg. 4.2), or one that uses double computation or sign-then-verify to detect fault injections (Alg. 4.3). Shuffling was simulated by applying the skipping fault to a randomly chosen coefficient. In contrast to the plain attack, the number of needed faults is not fixed here. The randomness in the target coefficient and the exclusion of ineffective faults leads to an overhead in the number of equations that are required in total for ending up with a solvable equation system for each component of $\mathbf{s_1}$ (cf. Section 4.3). To test the ineffective fault attack, the implementation sets the correction bound $b = 0$. The results for all tested modifications of the skipping fault correction attack can also be found in Table 1.

Additionally, for the skipping fault correction attack, we introduced the correction bound b as an additional parameter (cf. Section 4.2). It allows to balance the attack by limiting the search space for the value of $(c\mathbf{s_1})_j[i]$ to the most probable candidates around zero. As higher, unlikely values for $(c\mathbf{s_1})_j[i]$ might tend to correlate with unsuccessful skipping faults, the attacker can try to avoid those by lowering b. The influence of b on the number of faults is illustrated in Fig. 4: As expected, lowering b increases this number,

and the increase correlates to the distribution of the coefficients in $c\mathbf{s_1}$ (cf. Fig. 3).

For the experimental verification of the online phase resp. the fault injection for this attack we refer to the practical evaluations in [RJH+19], as the required faulty signatures are the same in their attack.

6.2 Correction Attack with Fault in A: Simulation

For our proposed attack on the matrix $\mathbf{A}$ (Section 5), we first tested the key recovery with simulated faults.

The faulty signatures, analogous to the skipping fault above, were generated by setting specific coefficients of $\hat{\mathbf{A}}$ to zero in `crypto_sign_signature` after it has been expanded (but before entering the rejection loop of the signature generation). For each parameter set, we generated ten different key pairs, and for each key pair, we computed $\ell \cdot n$ faulty signatures, one for each coefficient position in the first column of $\hat{\mathbf{A}}$.

For the offline part of the attack, we developed a highly parallelized program that makes use of the AVX2-optimized Dilithium implementation, specifically the verification, of the Dilithium authors [ABD+]. We ran the attack program on a server that features an AMD EPYC 7742 CPU clocked at 2.68 GHz and used all available 256 threads. For the simulated faults, one full key recovery takes on average (over the ten randomly generated keys)

- 3 minutes and 30 seconds for Dilithium2,
- 5 minutes and 19 seconds for Dilithium3, and
- 9 minutes and 53 seconds for Dilithium5.

6.3 Correction Attack with Fault in A: Practical Experiments

To verify that our assumptions regarding the fault model are valid, we also attacked a real device running Dilithium. The experiments were conducted using a ChipWhisperer Lite [Newa] connected to a ChipWhisperer UFO board with an STM32F405 target [Newb]. The device ran the optimized pqm4 implementation of Dilithium [KPR+] with one slight change: the memory for the matrix $\mathbf{A}$ was initialized with 0. The clock frequency was set to 24 MHz, which is the default for pqm4. Faults were induced through the clock-glitching capability of the ChipWhisperer board.

The targeted implementation expands the matrix $\mathbf{A}$ once at the start of every signature generation. In line with Dilithium's specification, it uses rejection sampling on the output of SHAKE called with the public seed ρ and the matrix indices (i, j). In this process, $\lceil \log_2 q \rceil$ bits are loaded from the SHAKE output, compared to q, and, if smaller, stored to the next coefficient of $\mathbf{A}$ (cf. Alg. 6.1).

To simplify analysis, we added a trigger signal notifying the start of the rejection-sampling procedure, i.e., after the call to SHAKE, of the currently targeted polynomial of $\mathbf{A}$. We experimentally determined reliable glitch parameters allowing us to target the store operation of a coefficient and computed the cycle offsets needed to target each individual index. The injected clock glitch caused the targeted coefficient to be zero for the rest of the signature generation while leaving all other coefficients undisturbed.

Our experiments showed a successful fault injection in 30 to 90% of all tests. We discovered a significantly lower success rate for coefficients with an index equal to 1 modulo 4 compared to the other coefficients. The pointer to the SHAKE output is advanced by 3 at a time, meaning that only every fourth load operation is word-aligned, which could explain this behavior. However, as described in the next section, our attack was successful with these fault injection probabilities. Hence, we refrained from further investigating the fault behavior.

Algorithm 6.1: ExpandA

Input: Public key part ρ

for every polynomial $\hat{a}_{i,j}$ in $\hat{\mathbf{A}}$ **do**
  $stream = \mathsf{SHAKE}(\rho|256 * i + j)$;
  while not every coefficient of $\hat{a}_{i,j}$ is set **do**
    take the next three bytes from *stream*;
    set highest bit to 0;
    if the resulting 23 bit number is smaller than q **then**
      use it as the next coefficient

return $\hat{\mathbf{A}} = (\hat{a}_{i,j})_{(i,j)\in\{0,\ldots,k-1\}\times\{0,\ldots,\ell-1\}}$

Dealing with Failed Fault Injections. As mentioned above, fault injection did not always have the desired result. We use several steps to deal with this fact. First, we disregard all signatures that verify without correction, which is the case either when the respective coefficient of $\hat{\mathbf{A}}$ was already zero or the fault simply did not work. Second, we discard signatures where the correction was not successful, i.e., where no value of $\Delta\hat{\mathbf{A}}\mathbf{y}$ leads to a satisfied verification condition. This can happen if the fault has affected more than one coefficient or the wrong position, which causes the coefficient recovery attempt to fail with overwhelming probability.

If the fault did in fact influence our intended target coefficient, but in some unintended manner leading to an unknown faulty value and thus an incorrect $\Delta\hat{\mathbf{A}}$, the correction procedure will still return a value for the targeted coefficient of $\hat{\mathbf{y}}/\hat{\mathbf{s}}_1$. However, the recovered value will be incorrect. Experimentally, we found that such incorrectly recovered values of $\hat{\mathbf{s}}_1$ take on random values, potentially due to using a randomized $\mathbf{y}$ and c in each signing operation. Consequently, we performed multiple faults per coefficient position during the online phase and repeated recovery until we obtained the same coefficient candidate twice for one position of $\hat{\mathbf{s}}_1$. For Dilithium2, this procedure successfully recovered 961 of 1 024 coefficients from 10 240 practically faulted signatures, i.e., 10 faults per key coefficient.[5] The offline search took 19 minutes of server time. For the remaining 61 coefficients, no two signatures yielded the same value for the specific $\hat{s_1}_{j_2}[i]$, thus we marked them as unknown.

To handle these unknown coefficients of $\hat{\mathbf{s}}_1$, we applied the lattice-reduction approach described in Section 5.2 (once per polynomial). Using the BKZ algorithm of the fplll library [dt23] with block size 5, we successfully recovered the secret $\mathbf{s_1}$ in about 20 minutes on a standard working laptop.

7 Conclusion

This work presented two new key-recovery fault attacks on Dilithium's signing procedure. Both methods can be used to attack the, in terms of implementation attacks, more robust (and now default) randomized signing mode, thereby demonstrating that simply switching away from deterministic signing does not lead to a sufficient level of fault robustness. In addition, the attacks can be extended such that some popular countermeasures, such as shuffling, can be circumvented. Finally, the fault on the public $\mathbf{A}$ demonstrates that fault susceptibility extends far beyond side-channel sensitive operations.

[5] This number could be drastically reduced by abandoning the strict differentiation between the online and offline phases, repeating the fault procedure for each position until the same value is recovered twice.

Acknowledgements

The work described in this paper has been supported by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under Germany's Excellence Strategy - EXC 2092 CASA - 390781972, by the German Federal Ministry of Education and Research BMBF through the project "MANNHEIM-FlexKI" (01IS22086I) and the project "Sec4IoMT" (16KIS1689), and by European Commission under the grant agreement number 101070374.

A Disturbing the Second Input of $\mathsf{UseHint}_q$

Lemma 1. *Let q and α be positive integers with $q > 2\alpha, q \equiv 1 \mod \alpha, \alpha$ even, let $\mathbf{r}, \mathbf{z}, \mathbf{x}$ be vectors with coefficients in $\mathcal{R}_q$ with $||\mathbf{z}||_\infty \leq \alpha/2$ and let $\mathbf{h}$ be a hint vector created by the Dilithium* sign *function. If*

$$\mathsf{UseHint}_q(\mathsf{MakeHint}_q(\mathbf{z}, \mathbf{r}), \mathbf{r}, \alpha) = \mathsf{UseHint}_q(\mathsf{MakeHint}_q(\mathbf{z}, \mathbf{r}), \mathbf{r} + \mathbf{x}, \alpha)$$

then for all but at most ω coefficients in $\mathbf{x}$ holds $|x_j^i| < \gamma_2$.

Proof. The definition of Dilithium's helper functions ([BDL+21]) shows that for all entries where $\mathbf{h} = 0$

$$\begin{aligned}&\mathsf{UseHint}_q(\mathsf{MakeHint}_q(\mathbf{z}, \mathbf{r}), \mathbf{r}, \alpha) = \mathsf{HighBits}_q(\mathbf{r}, \alpha), \text{ and}\\ &\mathsf{UseHint}_q(\mathsf{MakeHint}_q(\mathbf{z}, \mathbf{r}), \mathbf{r} + \mathbf{x}, \alpha) = \mathsf{HighBits}_q(\mathbf{r} + \mathbf{x}, \alpha).\end{aligned}$$

Together with the assumption it implies

$$\mathsf{HighBits}_q(\mathbf{r} + \mathbf{x}, \alpha) = \mathsf{HighBits}_q(\mathbf{r}, \alpha),$$

and that again means that, at those points, $\mathbf{x}$ has no influence on the high bits of $\mathbf{r} + \mathbf{x}$, and thus $|x_j^i| < \gamma_2$. The number of non-zero entries in $\mathbf{h}$ is guaranteed to be bounded by ω by the sign algorithm, which proofs the claim. □

References

[ABB+22] Jean-Philippe Aumasson, Daniel J. Bernstein, Ward Beullen, Christoph Dobraunig, Maria Eichlseder, Scott Fluhrer, Stefan-Lukas Gazdag, Andreas Hülsing, Panos Kampanakis, Stefan Kölbl, Tanja Lange, Martin M. Lauridsen, Florian Mendel, Ruben Niederhagen, Christian Rechberge, Joost Rijneveld, Peter Schwabe, and Bas Westerbaan. SPHINCS+: Submission to the NIST post-quantum project, v.3.1. Homepage, 2022. https://sphincs.org/data/sphincs+-r3.1-specification.pdf.

[ABC+23] Melissa Azouaoui, Olivier Bronchain, Gaëtan Cassiers, Clément Hoffmann, Yulia Kuzovkova, Joost Renes, Tobias Schneider, Markus Schönauer, François-Xavier Standaert, and Christine van Vredendaal. Protecting Dilithium against leakage revisited sensitivity analysis and improved implementations. *IACR TCHES*, 2023(4):58–79, 2023.

[ABD+] Avanzi, Bos, Ducas, Kiltz, Lepoint, Lyubashevsky, Schanck, Schwabe, Seiler, and Stehle. Reference implementation of Dilithium. git repository. https://github.com/pq-crystals/dilithium, as of May 3, 2023.

[AGVW17] Martin R. Albrecht, Florian Göpfert, Fernando Virdia, and Thomas Wunderer. Revisiting the expected cost of solving uSVP and applications to LWE. In *ASIACRYPT (1)*, volume 10624 of *Lecture Notes in Computer Science*, pages 297–322. Springer, 2017.

[BBK16] Nina Bindel, Johannes Buchmann, and Juliane Krämer. Lattice-based signature schemes and their sensitivity to fault attacks. In *FDTC*, pages 63–77. IEEE Computer Society, 2016.

[BDL+21] Shi Bai, Léo Ducas, Tancrède Lepoint, Vadim Lyubashevsky, Peter Schwabe, Gregor Seiler, and Damien Stehlé. CRYSTALS-Dilithium: Algorithm specifications and supporting documentation (version 3.1). Homepage, 2021. https://pq-crystals.org/dilithium/data/dilithium-specification-round3-20210208.pdf.

[BP18] Leon Groot Bruinderink and Peter Pessl. Differential fault attacks on deterministic lattice signatures. *IACR TCHES*, 2018(3):21–43, 2018.

[BVC+23] Alexandre Berzati, Andersson Calle Viera, Maya Chartouny, Steven Madec, Damien Vergnaud, and David Vigilant. Exploiting intermediate value leakage in Dilithium: A template-based approach. *IACR TCHES*, 2023(4):188–210, 2023.

[CGTZ23] Jean-Sébastien Coron, François Gérard, Matthias Trannoy, and Rina Zeitoun. Improved gadgets for the high-order masking of Dilithium. *IACR TCHES*, 2023(4):110–145, 2023.

[CKA+21] Zhaohui Chen, Emre Karabulut, Aydin Aysu, Yuan Ma, and Jiwu Jing. An efficient non-profiled side-channel attack on the CRYSTALS-Dilithium post-quantum signature. In *ICCD*, pages 583–590. IEEE, 2021.

[dt23] The FPLLL development team. fplll, a lattice reduction library, Version: 5.4.5. Available at https://github.com/fplll/fplll, 2023.

[EAB+23] Mohamed ElGhamrawy, Melissa Azouaoui, Olivier Bronchain, Joost Renes, Tobias Schneider, Markus Schönauer, Okan Seker, and Christine van Vredendaal. From MLWE to RLWE: A differential fault attack on randomized & deterministic Dilithium. *IACR TCHES*, 2023(4):262–286, 2023.

[EFGT18] Thomas Espitau, Pierre-Alain Fouque, Benoît Gérard, and Mehdi Tibouchi. Loop-abort faults on lattice-based signature schemes and key exchange protocols. *IEEE Trans. Computers*, 67(11):1535–1549, 2018.

[FHK+20] Pierre-Alain Fouque, Jeffrey Hoffstein, Paul Kirchner, Vadim Lyubashevsky, Thomas Pornin, Thomas Prest, Thomas Ricosset, Gregor Seiler, William Whyte, and Zhenfei Zhang. FALCON: Fast-fourier lattice-based compact signatures over NTRU. specification v1.2. Homepage, 2020. https://falcon-sign.info/falcon.pdf.

[GKS21] Denisa O. C. Greconici, Matthias J. Kannwischer, and Amber Sprenkels. Compact dilithium implementations on cortex-M3 and cortex-M4. *IACR TCHES*, 2021(1):1–24, 2021. https://tches.iacr.org/index.php/TCHES/article/view/8725.

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

[HP23] Daniel Heinz and Thomas Pöppelmann. Combined fault and DPA protection for lattice-based cryptography. *IEEE Trans. Computers*, 72(4):1055–1066, 2023.

[IMS+22] Saad Islam, Koksal Mus, Richa Singh, Patrick Schaumont, and Berk Sunar. Signature correction attack on Dilithium signature scheme. In *EuroS&P*, pages 647–663. IEEE, 2022.

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

[Lyu09] Vadim Lyubashevsky. Fiat-Shamir with aborts: Applications to lattice and factoring-based signatures. In *ASIACRYPT*, volume 5912 of *Lecture Notes in Computer Science*, pages 598–616. Springer, 2009.

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

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

[Newa] NewAE. CW1173 ChipWhisperer-Lite. https://rtfm.newae.com/Capture/ChipWhisperer-Lite.html.

[Newb] NewAE. CW308T-STM32F. https://rtfm.newae.com/Targets/UFO%20Targets/CW308T-STM32F.html.

[NIS22] NIST. Pqc standardization process: Announcing four candidates to be standardized, plus fourth round candidates. https://csrc.nist.gov/News/2022/pqc-candidates-to-be-standardized-and-round-4, 2022.

[NIS23] NIST Computer Security Division. FIPS 204 (Draft): Module-Lattice-Based Digital Signature Standard, 2023. https://csrc.nist.gov/pubs/fips/204/ipd.

[RCDB23] Prasanna Ravi, Anupam Chattopadhyay, Jan Pieter D'Anvers, and Anubhab Baksi. Side-channel and fault-injection attacks over lattice-based post-quantum schemes (Kyber, Dilithium): Survey and new results. *ACM TECS*, June 2023.

[RJH+19] Prasanna Ravi, Mahabir Prasad Jhanwar, James Howe, Anupam Chattopadhyay, and Shivam Bhasin. Exploiting determinism in lattice-based signatures: Practical fault attacks on pqm4 implementations of NIST candidates. In *AsiaCCS*, pages 427–440. ACM, 2019.

[RYB+23] Prasanna Ravi, Bolin Yang, Shivam Bhasin, Fan Zhang, and Anupam Chattopadhyay. Fiddling the twiddle constants - fault injection analysis of the Number Theoretic Transform. *IACR TCHES*, 2023(2):447–481, 2023.

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

[Wat87] William C Waterhouse. How often do determinants over finite fields vanish? *Discrete Mathematics*, 65(1):103–104, 1987.