Differential Fault Attacks on Deterministic Lattice Signatures

Leon Groot Bruinderink[1] and Peter Pessl[2]

[1] Technische Universiteit Eindhoven, The Netherlands, l.groot.bruinderink@tue.nl
[2] Graz University of Technology, Austria, peter.pessl@iaik.tugraz.at

Abstract. In this paper, we extend the applicability of differential fault attacks to lattice-based cryptography. We show how two deterministic lattice-based signature schemes, Dilithium and qTESLA, are vulnerable to such attacks. In particular, we demonstrate that single random faults can result in a nonce-reuse scenario which allows key recovery. We also expand this to fault-induced partial nonce-reuse attacks, which do not corrupt the validity of the computed signatures and thus are harder to detect.
Using linear algebra and lattice-basis reduction techniques, an attacker can extract one of the secret key elements after a successful fault injection. Some other parts of the key cannot be recovered, but we show that a tweaked signature algorithm can still successfully sign any message. We provide experimental verification of our attacks by performing clock glitching on an ARM Cortex-M4 microcontroller. In particular, we show that up to 65.2% of the execution time of Dilithium is vulnerable to an unprofiled attack, where a random fault is injected anywhere during the signing procedure and still leads to a successful key-recovery.

Keywords: Differential fault attacks · post-quantum cryptography · lattice-based cryptography · digital signatures

1 Introduction

Large-scale quantum computing is a major threat to currently used public-key cryptosystems. While it is uncertain when large-enough quantum computers will see the light of day, there is steady progress as, e.g., shown by the recent unveiling of a 72 qubit device [Kel18]. For this reason, the search for quantum-secure alternatives to discrete-logarithm and factoring-based solutions for public-key primitives is in full swing. This is demonstrated by the high interest in the NIST Post-Quantum Cryptography standardization process[1] [NIS]. In total 82 submissions were received at the very recent first-round deadline [Moo17]. This number trumps previous cryptographic competitions such as for AES and SHA-3 by far. A particularly interesting area of post-quantum research is lattice-based cryptography, as it appears to offer comparatively compact keys and ciphertexts as well as high computational performance. Possibly due to this reason, lattice-based cryptography is the largest category in terms of submissions.

*Author list in alphabetical order; see https://www.ams.org/profession/leaders/culture/CultureStatement04.pdf. This work was supported in part by the Commission of the European Communities through the Horizon 2020 program under project number 645622 (PQCRYPTO) and in part by the Austrian Research Promotion Agency (FFG) via the K-project DeSSnet, which is funded in the context of COMET – Competence Centers for Excellent Technologies by BMVIT, BMWFW, Styria and Carinthia. Date of this document: April 16, 2018

[1] NIST repeatedly stated that this process is not supposed to be a competition [Moo17].

There are ongoing discussions on the black-box security of these submissions[2]. In addition, secure implementations are another important aspect of the standardization process. The proposals should be easy to implement both correctly (ideally also misuse resistant) and securely. Proposals that offer such characteristics on a wide variety of platforms, including PCs as well as constrained devices like smart cards, are more desirable. Naturally this requires analysis of many implementation attacks, such as passive side-channel attacks (cf. [MOP07]) as well as active fault attacks (cf. [BCN+06]). The latter are a well-known threat to embedded devices. Rowhammer, a remote software-only fault attack [KDK+14, GMM16], demonstrated that also high-performance PCs are vulnerable. As implementations are evaluated in terms of both security and performance, they should be made resistant to such attacks with minimal costs.

In this regard, an interesting property of many lattice-based signature schemes is that they make use of the classic Fiat-Shamir transform, which allows constructing a digital signature scheme based on an interactive proof-of-knowledge protocol [FS86]. Concretely, two NIST submissions, qTESLA [BAA+17] and Dilithium [LDK+17], use a variant of the transform called Fiat-Shamir with Aborts [Lyu09]. However, signature schemes built using the Fiat-Shamir transform, such as the elliptic curve digital signature algorithm ECDSA, have a well-known caveat. They require that each message is signed with a unique nonce, a reuse leads to trivial key recovery. This requires proper implementations as such a nonce reuse can be caused, e.g., by using low-entropy seeds for PRNGs or by using a constant nonce. The latter mistake allowed the infamous attack on the PlayStation3 console [bms10]. In order to sidestep the problem of erroneous implementations and generating good seeds, the signature scheme can be made entirely deterministic. That is, signing the same message with the same key always leads to an identical signature. Both qTESLA and Dilithium use this approach and thus follow in the footsteps of proposals such as EdDSA [BDL+11] and deterministic ECDSA [Por13].

This solution, however, creates problems when it comes to fault attacks. An attacker can let a victim sign the same message twice, but introduce a computational fault in one of the signature computations. This results in different signatures using the same nonce and thus in a key recovery. In fact, recent work [BP16, ABF+17, PSS+17] explored the vulnerability of elliptic-curve signatures against such differential fault attacks, including Rowhammer-based ones [PSS+17].

The vulnerability of lattice-based deterministic signatures, however, is less clear. The possibility of such differential attacks was already hinted at [LDK+17, BAA+17], yet many questions remain open. Concretely, the abortion technique introduced by Lyubashevsky [Lyu09] and used by both qTESLA and Dilithium may hamper the attack. Furthermore, the different algebraic structure might open up new attack venues. Understanding the possibilities of such fault attacks is relevant in the standardization process and possible deployment of these schemes.

Our contributions. In this paper, we show the applicability of differential fault attacks on deterministic lattice-based signature schemes. We focus on Dilithium, but all our attacks apply to qTESLA as well. We explore how and where these schemes are vulnerable to single random faults and show how fault-induced nonce reuse allows extracting the secret key. Furthermore, we show attacks that can easily create and then efficiently exploit a *partial* nonce-reuse. This scenario yields valid signatures and thus allows to bypass some generic countermeasures.

In Dilithium and qTESLA, a unique signature vector $\mathbf{z} = \mathbf{y} + c\mathbf{s}$ is constructed out of a challenge c, a secret element $\mathbf{s}$, and a deterministically computed nonce $\mathbf{y}$. The attack is focused on faulting the computation of challenge c, leaving the nonce $\mathbf{y}$ untouched and thus creating a nonce reuse scenario. By carefully examining two signatures of the same

[2]Official forum at https://groups.google.com/a/list.nist.gov/forum/#!forum/pqc-forum

message yet with a (due to a fault) different challenge c, $\mathbf{s}$ can be extracted using linear algebra. We identify multiple operations inside the Dilithium signing algorithm that are vulnerable, i.e., where a random fault can lead to nonce reuse. We say "can", as the use of the Fiat-Shamir with Aborts framework leads to not all faults being exploitable. We determine the success probabilities for all fault scenarios, they range from 14 % to 91 %. In addition to these scenarios, we also explore fault-induced *partial* nonce reuse. There, the fault attack is specifically focused on the computation of nonce $\mathbf{y}$, but in such a way that *only a portion* of the computation is different. We exploit this by transforming key recovery into a unique shortest-vector problem, and show how to solve it using the BKZ lattice-reduction algorithm. While previous work already exploited such partial reuse scenarios for ECC [ABF+17], our attacks are much less restrictive regarding injected faults.

Successful extraction of $\mathbf{s}$ alone, however, does not directly allow to run the signing algorithm. This is due to Dilithium's public-key compression, which causes that some additional elements of the secret key cannot be computed from just $\mathbf{s}$. Thus, we show a tweaked signature algorithm that can still sign any new message despite lacking some parts of the key.

We verified the vulnerabilities by performing clock glitching on an ARM Cortex-M4 microcontroller. In particular, we induced random faults during polynomial multiplication and in the SHAKE extendable output function. We show that an attacker with detailed knowledge of the executed code can easily inject faults at correct locations despite some non-constant time behavior. Still, an unprofiled attacker who injects a fault anywhere during the signing process still has a high chance of succeeding. Up to 65.2 % of the execution time of Dilithium is vulnerable to our attacks.

We finally give a discussion on generic countermeasures against the attacks and reason about their applicability and implementation costs. We conclude that probably the simplest yet most effective countermeasure is a rerandomization of deterministic sampling, which, however, is not covered by the security proof of Dilithium.

Outline. In Section 2, we give the necessary background to understand the remainder of the paper. In Section 3, we explore the possibilities of differential fault attacks on Dilithium. In Section 4, we show how to modify the signature algorithm such that the secret key element extracted by our attacks suffices to compute valid signatures for any message. In Section 5 we verify the vulnerabilities with real experiments on an ARM Cortex-M4 microcontroller. In Section 6 we end the paper with a discussion on countermeasures.

2 Background

In this section, we introduce the necessary background on lattices and the Dilithium signature scheme. We also provide a summary of previous attacks on implementations of lattice-based cryptography.

2.1 Lattice-Based Cryptography

For any positive integer q, we define the polynomial ring $\mathcal{R}_q = \mathbb{Z}_q[x]/(x^n + 1)$. The elements in $\mathcal{R}$ are naturally represented as polynomials of degree less than n. For each polynomial $f \in \mathcal{R}_q$, we can define the corresponding vector of coefficients in $\mathbb{Z}_q$ as $\underline{f} = (f_0, f_1, \ldots, f_{n-1})$. Addition of polynomials $f + g$ corresponds to addition of their coefficient vectors. Multiplication of two polynomials $f \cdot g \bmod (x^n + 1)$ can be written in matrix-vector notation as $f \cdot g = \underline{g}\mathbf{F} = \underline{f}\mathbf{G}$, where $\mathbf{F}, \mathbf{G} \in \mathbb{Z}_q^{n \times n}$ are matrices, whose columns are nega-cyclic rotations of $\underline{f}, \underline{g}$.

Two hard problems underlying many lattice-based cryptography schemes are Ring-LWE/Ring-SIS, which are defined over the ring $\mathcal{R}_q$. Given a public key $(a, t) \in \mathcal{R}_q^2$, for

Algorithm 1 Dilithium Key Generation

Output: Keypair (pk, sk)
1: $\rho \leftarrow \{0,1\}^{256}, K \leftarrow \{0,1\}^{256}$
2: $(\mathbf{s}_1, \mathbf{s}_2) \leftarrow S_\eta^l \times S_\eta^k$
3: $\mathbf{A} \in \mathcal{R}_q^{k \times \ell} := \mathsf{ExpandA}(\rho)$
4: $\mathbf{t} := \mathbf{A}\mathbf{s}_1 + \mathbf{s}_2$
5: $(\mathbf{t}_1, \mathbf{t}_0) := \mathsf{Power2Round}_d(\mathbf{t})$
6: $tr \in \{0,1\}^{384} := \mathsf{CRH}(\rho || \mathbf{t}_1)$
7: **return** $(pk = (\rho, \mathbf{t}_1), sk = (\rho, K, tr, \mathbf{s}_1, \mathbf{s}_2, \mathbf{t}_0))$

Ring-LWE an attacker is asked to find short polynomials s_1, s_2 such that $t \equiv a \cdot s_1 + s_2 \bmod q$. Module-LWE / Module-SIS are generalizations of Ring-LWE/Ring-SIS, respectively. There the problems are defined over $\mathcal{R}_q^{k \times \ell}$ for some positive integers $k, \ell > 1$: given a matrix $\mathbf{A} \in \mathcal{R}_q^{k \times \ell}$ and a vector $\mathbf{t} \in \mathcal{R}_q^k$, find two short elements $\mathbf{s}_1 \in \mathcal{R}_q^\ell, \mathbf{s}_2 \in \mathcal{R}_q^k$ such that $\mathbf{t} \equiv \mathbf{A} \cdot \mathbf{s}_1 + \mathbf{s}_2 \bmod q$. For the attacks described in this paper, this means that an attacker needs to find multiple secret key elements.

Additional Notation. With $:=$ we denote deterministic assignments, with $\leftarrow$ we refer to uniform probabilistic sampling from some set. We define the ℓ_2 and ℓ_∞ norm for $w \in \mathcal{R}_q$ by $||w||_2 = \sqrt{\sum_{i=0}^{n-1} w_i^2}$ and $\|w\|_\infty = \max\{|w_0|, |w_1|, \ldots, |w_{n-1}|\}$, where all w_i are represented by an element in the interval $[-\frac{q-1}{2}, \frac{q-1}{2}]$. This definition can be naturally expanded to vectors of polynomials. S_η denotes the subset of $\mathcal{R}_q$ that includes all elements w that satisfy $\|w\|_\infty \leq \eta$.

2.2 Deterministic Lattice Signatures

We now describe the two deterministic lattice-based signature schemes Dilithium [LDK+17] and qTESLA [BAA+17], both of which were submitted to the NIST call. For design rationale, associated security proofs, and more details (e.g., on various subroutines) we refer to the respective submission documents.

Dilithium. In this work we focus mainly on Dilithium, which is why we give a more in-depth description of this scheme. Dilithium is based on the Module-LWE/SIS assumption. It operates over the fixed base ring $\mathcal{R}_q = \mathbb{Z}_q[x]/(x^{256} + 1), q = 8380417$ and allows for flexibility by allowing different module parameters (k, ℓ). This means that code used for arithmetic in $\mathcal{R}_q$ can be reused for any parameter set, which makes an adaptation to other security levels easier.

Key generation is given in Algorithm 1. First, two random seeds ρ, K, and two key elements $\mathbf{s}_1$, $\mathbf{s}_2$ are sampled. The function $\mathsf{ExpandA}$ deterministically expands the seed ρ into a matrix $\mathbf{A} \in \mathcal{R}_q^{k \times \ell}$ using the extendable-output function (XOF) $\mathsf{SHAKE128}$. This is done to minimize public and private key sizes as only ρ needs to be stored instead of the full $\mathbf{A}$. The public key $\mathbf{t} = \mathbf{A}\mathbf{s}_1 + \mathbf{s}_2$ is compressed by feeding it into the $\mathsf{Power2Round_q}$ function, which computes a pair $(\mathbf{t}_1, \mathbf{t}_0)$ such that $\mathbf{t} = \mathbf{t}_1 \cdot 2^d + \mathbf{t}_0$. Only the upper part $\mathbf{t}_1$ is published. The lower bits $\mathbf{t}_0$ and a hash of the public key $tr = \mathsf{CRH}(\rho || \mathbf{t}_1)$ are included in the private key sk. CRH is shorthand for Collision Resistant Hash, Dilithium uses $\mathsf{SHAKE256}$ with an output length of 384 bits.

Dilithium is based on the Fiat-Shamir with Aborts Framework [Lyu09]. Simply speaking, in this framework a signature σ is rejected and signing restarted if σ does not follow some fixed distribution. This rejection sampling statistically hides any secret information in the signature and thus provides the zero-knowledge property. The structure of rejection

Algorithm 2 Dilithium Sign (simplified[3])

Input: Message M, private key $sk = (\rho, K, tr, \mathbf{s}_1, \mathbf{s}_2, \mathbf{t}_0)$
Output: Signature $\sigma = (\mathbf{z}, \mathbf{h}, c)$
1: $\mathbf{A} \in \mathcal{R}_q^{k \times \ell} \coloneqq \mathsf{ExpandA}(\rho)$ ▷ fA_ρ, fA_E
2: $\mu \in \{0,1\}^{384} \coloneqq \mathsf{CRH}(tr||M)$
3: $\kappa \coloneqq 0, (\mathbf{z}, \mathbf{h}) \coloneqq \perp$
4: **while** $(\mathbf{z}, \mathbf{h}) = \perp$ **do**
5: $\quad \mathbf{y} \in S^l_{\gamma_1 - 1} \coloneqq \mathsf{DeterministicSample}(K||\mu||\kappa)$ ▷ fY
6: $\quad \mathbf{w} \coloneqq \mathbf{Ay}$ ▷ fW
7: $\quad \mathbf{w}_1 \coloneqq \mathsf{HighBits}(\mathbf{w})$
8: $\quad c \in B_{60} \coloneqq \mathsf{H}(\mu||\mathbf{w}_1)$ ▷ fH
9: $\quad \mathbf{z} \coloneqq \mathbf{y} + c\mathbf{s}_1$
10: $\quad \mathbf{h} \coloneqq \mathsf{MakeHint}(-c\mathbf{t}_0, \mathbf{w} - c\mathbf{s}_2 + c\mathbf{t}_0)$
11: $\quad (\mathbf{r}_1, \mathbf{r}_0) \coloneqq \mathsf{Decompose}(\mathbf{w} - c\mathbf{s}_2)$
12: $\quad$ **if** $\|\mathbf{z}\|_\infty \geq \gamma_1 - \beta$ or $\|\mathbf{r}_0\|_\infty \geq \gamma_2 - \beta$ or $\mathbf{r}_1 \neq \mathbf{w}_1$ **then** $(\mathbf{z}, \mathbf{h}) \coloneqq \perp$
13: $\quad \kappa \coloneqq \kappa + 1$
14: **return** $\sigma = (\mathbf{z}, \mathbf{h}, c)$

Algorithm 3 Dilithium Verify (simplified[3])

Input: Public key $pk = (\rho, \mathbf{t}_1)$, message M, signature $\sigma = (\mathbf{z}, \mathbf{h}, c)$
1: $\mathbf{A} \in R_q^{k \times \ell} \coloneqq \mathsf{ExpandA}(\rho)$
2: $\mu \in \{0,1\}^{384} \coloneqq \mathsf{CRH}(\mathsf{CRH}(\rho||\mathbf{t}_1)||M)$
3: $\mathbf{w}_1 \coloneqq \mathsf{UseHint}(\mathbf{h}, \mathbf{Az} - c\mathbf{t}_1)$
4: **accept** iff $c = \mathsf{H}(\mu||\mathbf{w}_1)$

sampling can be easily seen in Algorithm 2, which shows a slightly simplified[3] version of the Dilithium signature algorithm. The comments in Algorithm 2 refer to our attack scenarios and can be ignored for now.

Signature generation starts off by recomputing $\mathbf{A}$ and hashing the message M together with the hashed public key tr. The abort loop starts off by using the function $\mathsf{DeterministicSample}$ to generate the noise $\mathbf{y} \in S^\ell_{\gamma_1 - 1}$. The product $\mathbf{w} = \mathbf{Ay}$ is compressed to $\mathbf{w}_1$ using $\mathsf{HighBits}$. The hint $\mathbf{h}$ later allows the verifier to recompute this $\mathbf{w}_1$. The hash function H instantiates the random oracle needed in the proof. It returns a sparse ternary polynomial $c \in B_{60}$, i.e., a polynomial with Hamming weight 60 and all non-zero coefficients in ± 1. The function $\mathsf{Decompose}$ returns both $\mathsf{HighBits}$ and $\mathsf{LowBits}$ of its input. Finally, several checks are performed that determine if the current signature is accepted or rejected.

Note that all operations in Algorithm 2 are completely deterministic and thus generate a unique signature for message M[4]. This property is also used in the proof of Dilithium in the Quantum Random Oracle Model (QROM) [KLS17]. The proof does allow a non-deterministic version, albeit at the cost of tightness and a loss in security proportional to the number of distinct signatures an adversary can observe per message.

For completeness, we also provide a simplified version of the verification procedure (Algorithm 3). Throughout this paper we use the recommended Dilithium parameter set III shown in Table 1. It claims 128 bits of security against a Quantum adversary. Other parameter sets are given in Appendix A. They mainly differ in the used (k, ℓ), so our later attacks are possible for all proposed sets.

[3]Some additional checks and constant subroutine arguments are omitted.
[4]A previous Dilithium description [DLL+17] is probabilistic, but did not include a proof in the QROM.

Table 1: Dilithium Parameter Set III (recommended)

n	q	d	weight(c)	γ_1	γ_2	(k, ℓ)	η	β
256	8380417	14	60	523776	261888	(5, 4)	5	325

qTESLA. Structurally, the signature scheme qTESLA [BAA+17] is very similar to Dilithium. It also uses a variant of the Fiat-Shamir with Aborts framework and is deterministic. Unlike Dilithium, its proof in the QROM model [ABB+17] allows for a non-deterministic version as well (without losing tightness). The main difference however is that qTESLA is based on the Ring-LWE/SIS assumptions instead of the module counterparts. Thus, it operates on $\mathcal{R}_q = \mathbb{Z}_q[x]/(x^n + 1)$ (so $k = \ell = 1$) with $n \geq 1024$. We will later demonstrate our attacks on the example of Dilithium. Still, with some minor modifications they all also apply to qTESLA. We defer the description of qTESLA to Appendix B, where we also highlight the similarities to Dilithium.

2.3 The SHAKE Extendable Output Function

Dilithium makes heavy use of the SHAKE Extendable Output Function (XOF). It is also an important target of our fault attack, which is why we now very briefly describe it. SHAKE uses the sponge construction [BDPV07]. The sponge construction has two internal parameters r and c called the rate and the capacity, where the capacity is chosen such that the sponge construction meets a desired level of security. We call the internal state of the sponge x, consisting of $r + c$ bits, with all bits initialized to zero. The sponge starts with the *absorb* phase. Any input to the sponge function is first padded, using some injective padding function, resulting in $k \geq 1$ input blocks $m_1||m_2||\ldots||m_k$. These message blocks are then XORed with the first r bits of the state x, interleaved with applications of a permutation $f : \{0,1\}^{r+c} \to \{0,1\}^{r+c}$. In SHAKE, the Keccak-$f$ permutation is used. After all message blocks are processed, the *squeeze* phase starts. Depending on the desired output length, the function iteratively returns the first r bit blocks of the internal state x, interleaved with applications of the permutation f. In Figure 1, we show an example for three input blocks and three output blocks. Note that this construction allows for any number of input and output blocks.

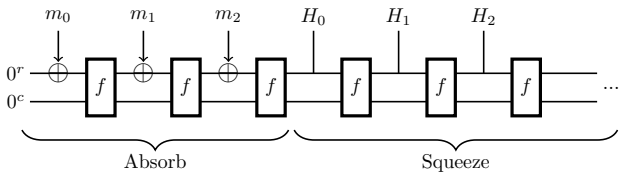

Figure 1: Sponge construction with three block input $m_0||m_1||m_2$ and three block output $H_0||H_1||H_2$. With $0^r, 0^c$ we denote the all-zero bit string of length r, c. The application of the padding function is not shown.

In the context of fault attacks, the important thing to note is that any manipulations in f corrupt the state x and thus affect all subsequent operations. For instance, faulting the first application of f in the squeeze phase leads to a correct H_0 but faulty $H_1, H_2, \ldots$.

2.4 Implementation Security of Lattice-Based Cryptography

Since lattice-based cryptography has gained traction in recent years, interest in its implementation-security aspect is also increasing. For the case of passive side-channel attacks, previous works showed, e.g., the applicability of cache attacks to lattice-based

signatures [GBHLY16, PBY17] and specialized power analysis of lattice-based cryptography [PPM17]. Countermeasures, such as masked implementations [OSPG16, RRVV15] as well as shuffling and other randomization techniques [Saa17], are also being proposed.

Many lattice-based schemes require high-precision sampling from a discrete Gaussian distribution. However, it appears to be difficult to implement discrete Gaussian samplers securely. In fact, previous attacks [GBHLY16, Pes16, PBY17, EFGT17] exploited the non-constant time nature of samplers. While there do exist first approaches to implementing discrete Gaussian samplers securely [MW17, HLS18, KRR+18], both Dilithium and qTESLA opt to use samples from the uniform distribution instead. This does not only allow much easier (constant-time) sampling, but also drastically simplifies the signature rejection step. In fact, non-constant time rejection was also exploited in previous work [EFGT17]. The downside of using the uniform distribution is slightly larger signatures. Compared to a Gaussian-based instantiation of Dilithium described in [DLL+17], signatures of the uniform version are larger by approximately 10 %.

Active implementation attacks on lattice-based cryptography also received some prior attention. Two concurrent works [BBK16, EFGT16] investigated fault attacks on non-deterministic lattice-based signature schemes, such as BLISS [DDLL13], GLP [GLP12], PASSSign [HPS+14], and ring-TESLA [ABB+16]. Espitau et al. [EFGT16] investigated loop-abort faults in the generation of the noise-polynomial $y \in \mathcal{R}_q$. This means that the sampling algorithm for this polynomial is cut off after the m'th noise coefficient, i.e. $\underline{y} = (y_0, \ldots, y_{m-1}, 0, \ldots, 0)$. A key-recovery is possible if $m \ll n$. The target distribution of y is not relevant; the general attack framework applies to both BLISS (which uses the discrete Gaussian distribution) and the other previously mentioned schemes (which all use the uniform distribution).

Like Dilithium and qTESLA, the above mentioned lattice-based signatures compute $z = y + cs$, where c, z are part of the signature σ and $s \in \mathcal{R}_q$ is a small secret key element. We can rewrite this equation as:

$$c^{-1}z \equiv c^{-1}y + s \bmod q \tag{1}$$

where we assume that $c \in \mathcal{R}_q$ is invertible (which is true with very high probability). As s is a small element, the target $t = c^{-1}z$ is close to a point in the lattice generated by the vectors $\{\underline{w}_i = c^{-1}x^i \bmod q \mid i \in \{0, \ldots, m-1\}\}$ and $q\mathbb{Z}^n$, and the difference is exactly s. This means that the closest-vector problem in (1) can be solved by, e.g., a lattice reduction followed by application of Babai's nearest plane algorithm. As this sub-lattice is of full dimension and too hard to solve at once, one can reduce the size of the problem to solve (1) for a subset $I \subseteq \{0, \ldots, n-1\}$ of indices, using the projection $\psi_I : \mathbb{Z}^n \to \mathbb{Z}^I$ given by $\psi_I((u_i)_{0 \le i < n}) = (u_i)_{i \in I}$. It can be shown that if the cardinality of any subset I is slightly larger than m (see the analysis in [EFGT16]), then (1) is solvable for subset I. By repeating this for multiple subsets, the complete secret key element s can be recovered. With knowledge of s the full secret key could be recovered using linear algebra.

2.5 Differential Fault Attacks on ECC

In this work we concentrate on differential fault attacks, in which the difference between a faulty and a correct output is used to determine information about the secret key. Previous work [BP16, ABF+17, PSS+17] explored such attacks on two deterministic elliptic curve signature schemes: EdDSA and deterministic ECDSA. Both of these signature schemes use the Fiat-Shamir transform, thus requiring the usage of a unique nonce per message. The fault attacks mainly focus on achieving nonce reuse, as this leads to a very efficient key-recovery.

Concretely, Poddebniak et al. [PSS+17] exploit the fact that the message is hashed twice in EdDSA. By manipulating the message in between these hashing operations with

Rowhammer, one can induce a nonce reuse and thus key recovery. Ambrose et al. [ABF+17] inspect a wider range of scenarios. They show that even random faults in certain operations can allow attacks. Additionally, they show that faults affecting the nonce itself are also usable. However, for this they require a very restrictive fault model. They need that the resulting error is limited to a few bits, as an exhaustive search is required to find the exact difference between the faulty and correct nonce.

3 Differential Faults on Deterministic Lattice Signatures

In this section, we present our differential fault attacks on Dilithium. As previously mentioned, these attacks apply to qTESLA as well, as we provide the attacks for general ℓ, k. First, we briefly describe our fault model. Then we explain the main intuition of our attacks. We identified multiple vulnerable operations, for each of them we finally describe how faulting can lead to key recovery. We also discuss additional properties, such as ease of fault injection, for the scenarios.

Fault Model. In this work we assume the possibility of injection a single random fault. These can encompass instruction skips, arithmetic faults, glitches in storage, and more. The faults are not restricted to specific operations but can be applied during a large section of execution time. This model is also used for some of the previously mentioned attacks on EdDSA [ABF+17] (some scenarios require a more restrictive fault model). In contrast, previous active attacks on lattice-based signatures required more control, such as the ability to abort a loop [EFGT16].

3.1 Intuition

The intuition behind our fault attacks is as follows. We let the signer sign the same message M twice. In the first invocation we do not inject any fault and receive a valid and proper signature $\sigma = (\mathbf{z}, \mathbf{h}, c)$. We inject a fault in the second run; we use $'$, e.g., $\mathbf{z}'$, to denote variables in this faulted invocation. More concretely, we inject a fault such that $\mathbf{y}'$ is undisturbed and due to the determinism equal to $\mathbf{y}$, yet $c' \neq c$ and thus $\mathbf{z}' = \mathbf{y} + c'\mathbf{s}_1$.

Thus, the fault induces a nonce-reuse scenario. When defining $\Delta\mathbf{z} = \mathbf{z} - \mathbf{z}'$ (and $\Delta c, \Delta\mathbf{y}$ analogously), we have $\Delta\mathbf{z} = \Delta\mathbf{y} + \Delta c \cdot \mathbf{s}_1 = \Delta c \cdot \mathbf{s}_1$. Thus, under the requirement that Δc is invertible, which is true with very high probability, then $\mathbf{s}_1 = \Delta c^{-1} \cdot \Delta\mathbf{z}$

The Fiat-Shamir with Abort structure, however, introduces an additional hurdle. We require that both the valid as well as the faulty signature computation terminate in the same iteration of the abortion loop. In other words, when using κ_f to denote the final value of the loop counter κ, we need that $\Delta\kappa_f = \kappa_f - \kappa'_f = 0$. Observe that in Algorithm 2, loop counter κ is input to DeterministicSample. Hence, to achieve $\mathbf{y} = \mathbf{y}'$ we have the requirement that $\Delta\kappa_f = 0$. Due to faulty intermediates and the influence of the rejection tests, this is obviously not guaranteed.

In the remainder of this section we discuss concrete fault scenarios. That is, we explain which operations in Algorithm 2 can be faulted such that key-recovery is possible. For each scenario we will give the exploitation technique as well as state its success probability, i.e., the chance that it terminates in the same loop iteration and thus $\Delta\kappa_f = 0$. This probability was estimated using at least 10 000 fault simulations per scenario. An overview of the scenarios is given in Table 2, they are listed in order of appearance in Algorithm 2. The order of description will be different.

Table 2: Fault scenarios discussed in this paper

Name	Section	Description
fA_ρ	3.4	Corrupt ρ during import of sk
fA_E	3.4	Random fault in expansion $\mathbf{A} \coloneqq \mathsf{ExpandA}(\rho)$
fY	3.5	Random fault in sampling $\mathbf{y} \coloneqq \mathsf{DeterministicSample}(\cdot)$
fW	3.3	Random fault in polynomial multiplication $\mathbf{w} \coloneqq \mathbf{Ay}$
fH	3.2	Random fault in call to H

3.2 Scenario: fH

Probably the most intuitive way to achieve a nonce-reuse is the fH scenario, where a random fault is injected into the computation $c \in B_{60} \coloneqq \mathsf{H}(\mu||\mathbf{w}_1)$. This can be achieved by either manipulating one of the inputs $\mu, \mathbf{w}_1$ immediately before they are being used in H, or by directly injecting a fault into the hash function H itself.

We will show in Section 5.1 that it is a very reasonable assumption that an attacker can inject a fault in the correct iteration κ_f, i.e., the last one in the non-faulty computation. If the rejection step is then passed with the different c', secret element $\mathbf{s}_1$ can be recovered as described in Section 3.1.

Since c is a sparse ternary polynomial and $\mathbf{s}_1 \in S_\eta^l$ has small coefficients, their product is also small. We depict its coefficient-wise probability distribution in Figure 2, it can be approximated with a (discretized) Gaussian distribution having zero mean and $\sigma \approx 24.3$. As $\|c\mathbf{s}\|_2 \ll \|\mathbf{y}\|_2, \|\mathbf{w}\|_2$, the rejection conditions for $\mathbf{z}$ and $\mathbf{r}_0$ are likely to hold for a different c as well. This results in a high success probability of over 90 %.

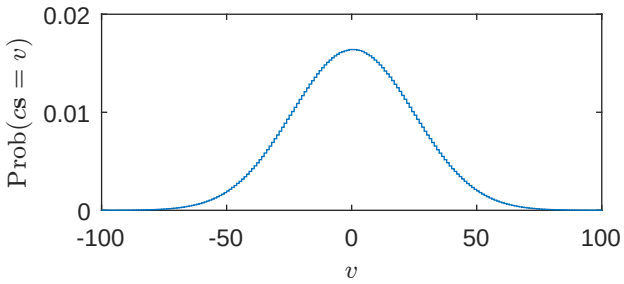

Figure 2: Coefficient-wise probability distribution of $c\mathbf{s}$

Determining Success. There are two ways to test if $\Delta\kappa_f = 0$ and thus key recovery is successful. The first method is to simply recover $\mathbf{s}_1$ and then test if it is small, i.e., $\mathbf{s}_1 \in S_\eta^l$. If $\Delta\mathbf{y} \neq 0$ then the recovered key will be a random vector in $\mathcal{R}_q^\ell$ which will not fulfill the bound on the ℓ_∞ norm. Alternatively, one can also exploit the small norm of $c\mathbf{s}$ by computing $\|\Delta\mathbf{z}\|_2$ (or also $\|\Delta\mathbf{z}\|_\infty$) and test if it is below a certain threshold. Again, $\Delta\mathbf{y} \neq 0$ will lead to a very large value of $\|\Delta\mathbf{z}\|_2$.

3.3 Scenario: fW

Instead of directly faulting the hash function H, it is also possible to alter $c \coloneqq \mathsf{H}(\mu||\mathbf{w}_1)$ by manipulating the computation of its inputs $\mu, \mathbf{w}_1$. The message / public key hash μ is also used as seed for $\mathsf{DeterministicSample}$, hence faults in the computation $\mu \coloneqq \mathsf{CRH}(tr||M)$ are not exploitable.

Faults in the computation of $\mathbf{w} := \mathbf{Ay}$ which lead to an incorrect $\mathbf{w}_1$, however, can be exploited. The required polynomial multiplications in $\mathcal{R}_q$ can be efficiently implemented using the Number Theoretic Transform (NTT). Still, the runtime of multiplication is higher than that of hashing, thus it can be a more viable target for fault attacks. An NTT is essentially an FFT-like transform over a prime field and uses similar implementation techniques, i.e., butterfly networks. Due to these techniques, the number of coefficients in $\mathbf{w}$ affected by a single random fault can range from 1 to all $n \cdot k$.

As unaffected coefficients of $\mathbf{w}$ clearly pass rejection and a single altered one is sufficient to achieve $\Delta c \neq 0$, minimizing the number of faulty coefficients increases the success probability. Thus, unlike in our other scenarios the concrete fault position has a much stronger impact. To give a sense of possible success probabilities, we evaluated the two most extreme cases. First, we inject a fault in the forward-NTT of $\mathbf{y}$. Such a fault spreads to all $n \cdot k$ coefficients of $\mathbf{w}$ and thus leads to a low success probability (25.3 %). Second, we fault the inverse-NTT applied to $\mathbf{w}$ such that only two coefficients are affected. With a success probability of over 90 %, this sub-scenario is similar to directly faulting H. Note that while single-coefficient faults are also possible, they are slightly less likely to lead to a successful key-recovery. This is due to the chance that a faulty coefficient w' still rounds to the correct $w'_1 = w_1$, which results in $\Delta c = 0$ and the fault not being exploitable.

3.4 Scenarios: fA_ρ, fA_E

Another possibility to achieve a faulty $\mathbf{w} = \mathbf{Ay}$ is to manipulate the expansion of seed ρ into the matrix $\mathbf{A}$. As seen in Algorithm 2, this is done before entering the abort loop and is thus always executed at the same time. Furthermore, $\mathsf{ExpandA}$ is a major contributor to overall runtime (cf. Section 5.2). Both these properties drastically simplify fault injection for this scenario. Also, $\mathbf{A}$ has a large memory footprint (20 kB in Dilithium-III) and is kept in storage for the entirety of signing. This makes it a particularly interesting target for memory-based faults, such as Rowhammer.

When focusing on more traditional faulting techniques, then differences in $\mathbf{A}$ can be achieved by either manipulating the seed ρ, e.g., during loading of the private key (scenario fA_ρ), or by inserting a glitch into the expansion $\mathbf{A} \in \mathcal{R}_q^{k \times \ell} := \mathsf{ExpandA}(\rho)$ (scenario fA_E). On first glance these scenarios might seem identical. There are, however, some major differences. Observe that in Algorithm 4, which sketches the method for expanding ρ into $\mathbf{A}$, the $k \cdot \ell$ polynomials comprising $\mathbf{A}$ are generated using independent calls to SHAKE. Thus, any single fault in the SHAKE permutation leads to just one corrupted polynomial. Consequently, after the matrix-vector multiplication $\mathbf{Ay}$ we have n differing coefficients of $\mathbf{w}$. This leads to a success probability of approximately 54 %.

Directly faulting ρ, either during import or in storage, obviously results in an all different $\mathbf{A}$ and thus $\mathbf{w}$. This decreases the success probability to just 14 %. However, this type of fault has a major advantage when it comes to defeating countermeasures. It is potentially (semi-)permanent and can thus, at least under certain circumstances, not be detected by the generic double-computation or verification-after-sign countermeasures. In Section 6 we discuss this in more detail.

3.5 Scenario: fY

So far, we have only discussed fault-induced nonce-reuse scenarios, i.e. the case where $\mathbf{y}' = \mathbf{y}$. For our final scenario, we will switch to *partial* nonce-reuse. Unlike all previous scenarios, inducing a partial reuse still leads to valid signatures and is thus not detectable with a signature verification.

We introduce some additional notation for element $\mathbf{t} \in \mathcal{R}_q^\ell$: we define $t_u \in \mathcal{R}_q$ to be the u'th element of $\mathbf{t}$ and $(\underline{t}_u)_v \in [-\frac{q-1}{2}, \frac{q-1}{2}]$ to be its v'th coefficient, for $0 \leq u < \ell$ and

Algorithm 4 ExpandA (ρ)

Input: Seed ρ
Output: uniform $\mathbf{A} \in \mathcal{R}_q^{k \times \ell}$

for $i \coloneqq 0 \dots k-1$ **do**
  for $j \coloneqq 0 \dots \ell - 1$ **do**
    $\mathbf{A}_{i,j} \coloneqq \mathsf{SamplePoly}(\rho || i || j, q)$
return A

function SamplePoly(s)
  $t \in \{0,1\}^{5 \cdot \text{SHAKErate}} \coloneqq \mathsf{SHAKE128}(s)$
  $u \coloneqq 0$
  while $u < n$ **do**
    $v \coloneqq$ next $\lceil \log_2 q \rceil$ bits of t
    if $v < q$ **then**
      $a_i \coloneqq v$
      $u \coloneqq u + 1$
  return a

$0 \leq v < n$. We define $\underline{e}_j$ for $0 \leq j < n$ to be the j-th unit vector, i.e. the vector with a 1 at position j and zero otherwise.

Simple Example. First, let us assume the following. We inject a fault in $\mathbf{y}' \in S^{\ell}_{\gamma_1 - 1}$ such that only a single coefficient $(\underline{y}'_u)_v \in \mathbf{y}$ (with index $u \in \{0, \dots, \ell - 1\}$ and $v \in \{0, \dots, n-1\}$) is changed to a random value (while preserving $|(\underline{y}'_u)_v| \leq \gamma_1 - 1$). Still, this leads to a completely different $\mathbf{w}_1$, and therefore to a different c' and $\mathbf{z}' = \mathbf{y}' + c'\mathbf{s}_1$.

We then compute $\mathbf{s}_1 = \Delta c^{-1} \cdot \Delta \mathbf{z}$ and determine u by simply using the one index for which $s_{1,u} \notin S_\eta$. For all $i \neq u$ we have that $\Delta y_i = 0$, thus recovery of these key polynomials succeeds. If we now compute the difference Δz_u, we will notice the following: for indices $i \neq v$ we see $|\Delta z_i| \leq 2\delta$ for some threshold δ chosen such that $\|c\mathbf{s}_1\|_\infty \leq \delta$ holds for any c and $\mathbf{s}_1$. Concretely, we can set $\delta = 60\eta$. The injected fault in $(\underline{y}'_u)_v$ can cause any difference to the value, but on average it will be large. On expectation $|(\underline{y}'_u)_v - (\underline{y}_u)_v|$ will be $\frac{2\gamma_1 - 1}{3}$, as both of these coefficients are random values in $[-(\gamma_1 - 1), (\gamma_1 - 1)]$ (by our assumption). Since $\|c\mathbf{s}_1\|_\infty \leq \delta \ll \frac{2\gamma_1 - 1}{3}$, we can detect index v and thus the position of the fault by using the index of maximum $|\Delta z_u|$.

We finally recover $s_{1,u}$ as follows. Simply speaking, we eliminate row v of the linear system $s_{1,u} = \Delta c^{-1} \cdot \Delta z_u$, guess the value of $(\underline{s}_{1,u})_v$ (exhaustive search), solve for the full $s_{1,u}$ and test if it is in S_η. Note that similarly we could also directly guess the value of $(\Delta \underline{y}_u)_v$ (instead of $(\underline{s}_u)_v$), albeit there the search-space is much larger. This latter scenario is the direct counterpart to the partial nonce reuse fault attack on elliptic curve signatures (as described in [ABF+17] and mentioned in Section 2.5): an exhaustive search is used to determine the exact error in the faulted nonce.

We will show next that for lattice-based signatures these partial-reuse attacks are way more powerful. The exhaustive search can be replaced with solving a lattice problem, which is much more efficient. The far larger number of tolerable errors allows replacing the very restrictive fault model (influencing a small number of bits of the nonce) with random faults in SHAKE.

Efficient Partial Nonce Reuse Attack. The nonce $\mathbf{y} \in S^{\ell}_{\gamma_1 - 1}$ is generated by function DeterministicSample, a simplified[5] version is given in Algorithm 5. Note that input seed

[5] For example, with very small probability the $5 \cdot$ SHAKErate bits are not enough to generate enough values for any y_i. In that case, another call to SHAKE and more rejection sampling is done.

Algorithm 5 $\mathsf{DeterministicSample}_{\gamma_1-1}(s)$ (simplified[5])

Input: Seed s
Output: $\mathbf{y} \in S^{\ell}_{\gamma_1-1}$
1: **for** $u := 0 \dots \ell - 1$ **do** ▷ Sample ℓ nonce polynomials
2: $\quad t \in \{0,1\}^{5 \cdot \text{SHAKErate}} := \mathsf{SHAKE256}(s||u)$
3: $\quad v := 0$
4: $\quad$ **while** $v < n$ **do** ▷ Rejection sampling
5: $\quad\quad r :=$ next $2\lceil \log_2 \gamma_1 \rceil$ bits of t
6: $\quad\quad$ **if** $r \leq 2(\gamma_1 - 1)$ **then**
7: $\quad\quad\quad (\underline{y}_u)_v := q + \gamma_1 - 1 - r$
8: $\quad\quad\quad v := v + 1$
9: **return a**

s changes whenever a signature is rejected (Algorithm 2), and the counter u will change the individual elements of $\mathbf{y}$. The idea is that we now fault SHAKE (Line 2), but in such a way that it only changes a few coefficients of y_u for some $u \in \{0, \dots, \ell - 1\}$. Since all coefficients of $(\underline{y}_u)$ are sampled sequentially, a fault that only affects the last few bytes of t' will only change the last few coefficients of $(\underline{y}_u)$.

As mentioned in Section 2.3, SHAKE operates on a state x of $r + c$ bits and consists of an absorb phase and a squeeze phase. If a fault is injected during the absorb phase, the output of SHAKE will be completely different. However, if the fault is injected near the end of the squeeze phase, only the last few applications of f will operate on a faulty state x and thus return an erroneous output (cf. Section 2.3). In particular, as DeterministicSample requests 5 output blocks of SHAKE (Line 2), an injected fault in, e.g., the last or second-to-last application of Keccak-f during the squeeze phase will cause changes in the last few bytes of t'. Thus, only the last few coefficients of y'_u will differ i.e. $\Delta y_u = (0, \dots, 0, \zeta_v, \zeta_{v+1}, \dots, \zeta_{n-1})$ for some index $v \in \{0, \dots, n-1\}$. Note that the index v for which the values start to differ will vary depending on how many elements were accepted from the first few output blocks of SHAKE. We can again detect index v similarly as mentioned previous: by taking the first index where $|\Delta z_u| \geq 2\delta$. However, we cannot apply the brute-force search for the corresponding elements in $(\underline{s}_{1,u})_{v \leq i < n}$ anymore: the search-space will be too large.

Instead, we will transform the search for these $n - v$ secret coefficients to a lattice problem similarly as described in Section 2.4. Thus, when writing:

$$t = \Delta c^{-1} \Delta z_u = \Delta c^{-1} \Delta y_u + s_{1,u}$$

we have a target t and want to determine the closest point on the lattice generated by Δc^{-1}. The difference between t and its closest lattice point is exactly $s_{1,u}$. Solving this closest-vector problem is made possible by using that the first v coefficients of Δy_u are 0. We use the lattice generated by basis vectors $\{\underline{w}_i = \Delta c^{-1} x^i \bmod q \mid i \in \{v, v+1, \dots, n-1\}\}$ and $q\mathbb{Z}^n$. Take $I = \{m, m+1, \dots, n-1\}$ to be the target subset of indices, where $m < v$ and apply ψ_I to these basis vectors, where ψ_I as defined in Section 2.4. We then cast the problem at hand to a unique shortest-vector problem as, e.g., described by Albrecht et al. [AFG13], and then apply a lattice-reduction algorithm (like LLL or BKZ). If successful, we retrieve a small $n - m$ dimensional vector $\underline{s}^{\text{guess}}$, whose coefficients correspond to the last $n - m$ coefficients of $\underline{s}_{1,u}$. To get the full $s_{1,u}$, we replace the last $n - m$ coefficients of $\Delta \underline{z}_u$ by the coefficients of $\underline{s}^{\text{guess}}$, transform rotation-matrix $\Delta\mathbf{C}$ of Δc into $\overline{\mathbf{C}}$ by replacing the last $n - m$ columns by the identity columns $\underline{e}_m, \underline{e}_{m+1}, \dots, \underline{e}_{n-1}$ and compute the full $\underline{s}^{\text{guess}} = \Delta \underline{z} \overline{\mathbf{C}}^{-1}$. We can verify correctness by checking that $\underline{s}^{\text{guess}} \in S_\eta$.

In our experiments, we injected a random fault in the last (denoted by 1P) or second-to-last (denoted by 2P) application of Keccak-f inside SHAKE (called in Algorithm 5, line 2).

Table 3: Results of injecting faults in DeterministicSample

Squeeze phase iteration	Average number of errors in y_u	Average running time lattice reduction
1P (last)	39	2.3s
2P (second to last)	93	35.8s

Table 4: Fault-attack success probability in percent

fA_ρ	fA_E	fY-1P	fY-2P	fW	fH
14.3	54.4	24.8	23.9	25.4 - 90.3	91.0

Since the input to SHAKE is shorter than the rate r, out of the total five applications of Keccak-f these are the fourth (2P) and fifth (1P), respectively. Faults in the 3 earlier applications of Keccak-f did not yield a solvable lattice problem. We performed 1000 experiments for both 1P and 2P and determined the average number of errors (so $n - v$) and the average running time for BKZ (on an Intel Xeon E5-4669 v4 @ 2.20GHz). In our experiments, we took m such that the cardinality of I is about $1.4(n - v)$. For the lattice reduction, we used BKZ with block-size 25 but included an early abort, i.e., we abort reduction as soon as a potential key-candidate (a vector in S_η) is found. The results are shown in Table 3. The success probability of the lattice reduction was 100%. Thus, if a fault is correctly injected and $\Delta\kappa_f = 0$, then the key $\mathbf{s}_1$ can always be recovered. The probability that $\Delta\kappa_f = 0$ is between 24 and 25 % (Table 4).

3.6 Summary of Scenarios

We now give a summary of the different fault scenarios. In Table 4 we restate the success probability of all fault scenarios. Recall that in scenario fW a large number of outcomes is possible, but we analyzed the best and worst possible outcomes. For scenarios fY, fH, and fW we assume that the fault is injected in the last iteration κ_f.

fH is the most intuitive scenario and also achieves the highest success probability. However, it is also the smallest of all targets (cf. Section 5.2). The lowest success probability is achieved for fA_ρ, yet with the huge advantage of being potentially permanent. Faulting the expansion of $\mathbf{A}$ offers both a large and fixed-time target. Finally, scenario fY lead to valid yet still exploitable signatures.

4 Signing with the Recovered Key

In the previous sections we showed how to recover $\mathbf{s}_1$ after a successful fault injection. However, $\mathbf{s}_1$ is only one component of the private key $sk = (\rho, K, tr, \mathbf{s}_1, \mathbf{s}_2, \mathbf{t}_0)$. The seed ρ, which is used for generating the matrix $\mathbf{A}$, is also part of the public key. tr can be trivially recomputed as $\mathsf{CRH}(\rho||\mathbf{t}_1)$ (cf. Algorithm 1). K is used as a secret input to the deterministic sampler and cannot be recovered with our attack. However, an attacker can just choose any random K and still produce valid signatures. The only downside here is that the owner of the full private key can test whether or not a signature is forged. He simply runs the signature algorithm and tests for equivalence, a new K will obviously result in a different yet still valid signature.

The situation for the two remaining components, namely $\mathbf{s}_2$ and $\mathbf{t}_0$, is less clear. Recall that $\mathbf{t} \coloneqq \mathbf{A}\mathbf{s}_1 + \mathbf{s}_2$ (cf. Algorithm 1). If $\mathbf{t}$ is known, then recovering $\mathbf{s}_2$ boils down to simple linear algebra. However, for compression one computes a pair $(\mathbf{t}_1, \mathbf{t}_0)$ satisfying

$\mathbf{t}_1 \cdot 2^d + \mathbf{t}_0 = \mathbf{t}$ and includes only the upper part $\mathbf{t}_1$ in the public key. Thus, the equation $\mathbf{t}_1 \cdot 2^d + \mathbf{t}_0 = \mathbf{A}\mathbf{s}_1 + \mathbf{s}_2$ cannot be directly solved.

Note also that during signature computation $\mathbf{s}_2$ and $\mathbf{t}_0$ are only used for hint generation and rejection purposes. Thus, there are no simple equations that can be exploited for recovering this part of the private key. This obviously does not imply that there is no information on $\mathbf{s}_2$ present. For instance, in a valid signature we have that $\|\mathbf{r}_0\|_\infty < \gamma_2 - \beta$, with $(\mathbf{r}_1, \mathbf{r}_0) := \mathsf{Decompose}(\mathbf{w} - c\mathbf{s}_2)$. As $\mathbf{w}$ is recoverable since $\mathbf{s}_1$ is already known, an attacker will get constraints for the possible values for $\mathbf{s}_2$. A large number of such constraints could result in a fully determined $\mathbf{s}_2$. However, we expect that a very large number of valid signatures and high computational effort is needed to perform such a recovery.

Instead, we now present a modified signing procedure (Algorithm 6) that does not require knowledge of $\mathbf{s}_2$. Thus, the property that only a single valid/faulty signature pair is needed for the attack is preserved. Algorithm 6 starts off by recomputing tr and sampling a random K, as described earlier. Then we compute $\mathbf{u} := \mathbf{A}\mathbf{s}_1 - \mathbf{t}_1 \cdot 2^d$, which is exactly the difference of the unknown quantities, i.e., $\mathbf{u} = \mathbf{t}_0 - \mathbf{s}_2$. Signature generation then continues as usual up until the computation of the hint $\mathbf{h}$.

In the original signing algorithm we have $\mathbf{h} := \mathsf{MakeHint}(-c\mathbf{t}_0, \mathbf{w} - c\mathbf{s}_2 + c\mathbf{t}_0)$. The second argument to $\mathsf{MakeHint}$ can be trivially rewritten as $\mathbf{w} - c\mathbf{s}_2 + c\mathbf{t}_0 = \mathbf{w} + c\mathbf{u}$. The first argument $-c\mathbf{t}_0$ cannot be computed without knowledge of $\mathbf{t}_0$. We get around this by exploiting the fact that $\mathbf{t}_0$ is vastly larger than $\mathbf{s}_2$, with coefficients in the intervals $[\pm 2^{d-1}]$ and $[\pm\eta]$, respectively. Thus, we have that $\mathbf{u} = \mathbf{t}_0 - \mathbf{s}_0 \approx \mathbf{t}_0$ and simply substitute $-c\mathbf{t}_0$ with $-c\mathbf{u}$.

We then skip all rejection conditions that cannot be tested without knowing $\mathbf{s}_2$ or $\mathbf{t}_0$. Essentially, we just test that if $\|\mathbf{z}\|_\infty \geq \gamma_1 - \beta$ and reject the signature if this is the case. Finally, we perform a verification of the signature to catch the very improbable case that $\mathsf{MakeHint}(-c\mathbf{u}, \mathbf{w} + c\mathbf{u}) \neq \mathsf{MakeHint}(-c\mathbf{t}_0, \mathbf{w} + c\mathbf{u})$.

Due to the removal of rejection conditions, this modified signing algorithm potentially leaks secret information. Thus, anyone being aware of the fact that signatures are computed by our modified algorithm could maybe also recover the secret key. Since all produced signatures are valid, there is no trivial way to test for this condition (without already knowing the key, as explained earlier).

5 Experimental Verification

In this section, we back up our previous theoretical expositions and simulations by running our attack on an actual device. After discussing our platform, we show how an attacker can inject a fault in the iteration κ_f without determining the concrete value. This requires at least some knowledge of the implementation. For this reason, we also demonstrate that a random fault anywhere during the signing procedure has a high chance of being exploitable.

Platform. For our experiments, we use an STM32F405 microcontroller (ARM Cortex-M4F) running on a ChipWhisperer CW308 side-channel evaluation board. We run the Dilithium C reference implementation[6] (compiled with `-O3`) and clock our device at 30 MHz. For attack evaluation, we signal the start and end of signing with a trigger pin. As faulting method we make use of clock glitches.

We mounted attacks for all scenarios except fA_ρ, all with success. For the scenarios targeting the SHAKE XOF, i.e., fA_E, fY, and fH, the ability to precisely time clock glitches and thus to attack very specific instructions is not needed. A single such permutation

[6] Reference implementation available at https://pq-crystals.org/dilithium/software.shtml

Algorithm 6 Dilithium Sign with Recovered Key $\mathbf{s}_1$

Input: Message M, private key part $\mathbf{s}_1$, public key $pk = (\rho, \mathbf{t}_1)$
Output: Signature $\sigma = (\mathbf{z}, \mathbf{h}, c)$

1: $tr \in \{0,1\}^{384} \coloneqq \mathsf{CRH}(\rho || \mathbf{t}_1)$ ▷ Recompute tr from public information
2: $K \leftarrow \{0,1\}^{256}$ ▷ Sample a random seed
3: $\mathbf{u} \coloneqq \mathbf{A}\mathbf{s}_1 - \mathbf{t}_1 \cdot 2^d$ ▷ $\mathbf{A}\mathbf{s}_1 - \mathbf{t}_1 \cdot 2^d = \mathbf{t}_0 - \mathbf{s}_2$
4: $\mathbf{A} \in \mathcal{R}_q^{k \times \ell} \coloneqq \mathsf{ExpandA}(\rho)$
5: $\mu \in \{0,1\}^{384} \coloneqq \mathsf{CRH}(tr || M)$
6: $\kappa \coloneqq 0, (\mathbf{z}, \mathbf{h}) \coloneqq \perp$
7: **while** $(\mathbf{z}, \mathbf{h}) = \perp$ **do**
8: $\quad \mathbf{y} \in S^l_{\gamma_1 - 1} \coloneqq \mathsf{DeterministicSample}(K || \mu || \kappa)$
9: $\quad \mathbf{w} \coloneqq \mathbf{A}\mathbf{y}$
10: $\quad \mathbf{w}_1 \coloneqq \mathsf{HighBits}(\mathbf{w})$
11: $\quad c \in B_{60} \coloneqq \mathsf{H}(\mu || \mathbf{w}_1)$
12: $\quad \mathbf{z} \coloneqq \mathbf{y} + c\mathbf{s}_1$
13: $\quad \mathbf{h} \coloneqq \mathsf{MakeHint}(-c\mathbf{u}, \mathbf{w} + c\mathbf{u})$ ▷ $\mathsf{MakeHint}(-\mathbf{c}(\mathbf{s}_2 - \mathbf{t}_0,), \mathbf{w} - c\mathbf{s}_2 + c\mathbf{t}_0)$
14: $\quad$ **if** $\|\mathbf{z}\|_\infty \geq \gamma_1 - \beta$ **then** ▷ Remove rejection conditions
15: $\quad\quad (\mathbf{z}, \mathbf{h}) \coloneqq \perp$
16: $\quad$ **else**
17: $\quad\quad$ **if** not $\mathsf{Verify}(pk, M, (\mathbf{z}, \mathbf{h}, c))$ **then** $(\mathbf{z}, \mathbf{h}) \coloneqq \perp$ ▷ Test for correctness
18: $\quad \kappa \coloneqq \kappa + 1$
19: **return** $\sigma = (\mathbf{z}, \mathbf{h}, c)$

takes approximately 40 000 clock cycles and we only require that its output is different, thus any random fault suffices. In fact, we did not determine the exact location or effect of the fault. Attacks on the polynomial multiplication (scenario fW) can benefit from more precise fault injection (see Section 3.3). However, even random faults yield a high success rate (Section 5.2).

5.1 Injecting a Fault in the Correct Iteration

Recall that a fault is only exploitable if both the faulted and the non-faulted execution of the signing algorithm terminate in the same iteration of the abort loop, i.e., $\Delta\kappa_f = 0$. Clearly, in the scenarios fY, fW, and fH, an attacker can maximize the success probability by injecting the fault in this last iteration κ_f.

The Dilithium reference implementation is constant (read: key-independent) time. The individual rejection conditions (line 12 of Algorithm 2) are still tested as soon as possible. This minimizes the runtime of failed iterations but does not leak sensitive information on the key. Quite on the contrary, this non-constant-time behavior somewhat complicates the fault attack. Even an attacker knowing κ_f cannot exactly pinpoint the time of execution of vulnerable operations and thus the best time to inject a fault.

We get around this by using the observation that the last loop iteration κ_f is, unlike the previous ones, constant time. Only there all operations are guaranteed to be performed and apart from the rejections the code is constant time. Thus, we determine the time of execution of vulnerable operations as follows. First, we perform the undisturbed signing and measure its runtime. And second, we simply subtract a fixed offset (depending on the to-be-faulted operation) from this overall runtime. We used this method for our attacks in the scenarios fY, fH, and fW, and were successful for any κ_f.

Table 5: Runtime-percentage of vulnerable code

	$\mathrm{fA_E}$	fY	fW	fH	**Sum**
$\kappa_f = 1$	47.4	3.8	11.2	2.9	**65.2**
Overall	24.3	2.0	5.7	1.5	**33.5**

5.2 Unprofiled Attacks

The above method is highly accurate, yet requires some device/code profiling. Concretely, an attacker needs to determine the time offsets (either from the start or finish of the signing operation) of the vulnerable code. This might not always be a realistic assumption. For this reason, we now show that an attacker injecting a random fault anywhere in the signing process still has a high chance of succeeding. We do so by measuring the runtime (in cycles) of the vulnerable code and relating it to the overall execution time (Table 5).

In the best-case scenario for such an attacker, the signing algorithm terminates in the first iteration ($\kappa_f = 1$). In this case, 65.2 % of execution time are vulnerable. In the general case (no restriction to $\kappa_f = 1$), the success probability goes down to one-third of the total execution time.

In both cases, sampling of the matrix $\mathbf{A}$ takes by far the most time. Additionally, it is performed at a fixed time in the execution, shortly after the invocation of the signing algorithm. Thus, in reality an unprofiled attacker faulting somewhere in this region has a much higher chance of hitting ExpandA than stated in Table 5.

In Figure 3, we further visualize the general case and compare runtime to success probability for different scenarios. Recall that depending on the concrete fault position, the success probability of scenario fW varies drastically (see Table 4). For the case of the unprofiled attacker, we narrowed down this probability by performing 1000 fault attacks on our target device, with faults at random positions inside fW. Approximately 62 % of these faults were exploitable. Faulting the call to H yields the highest success probability (Table 4), but also has the smallest footprint. As discussed in Section 3.5, 2/5 of the time spent on the SHAKE call by DeterministicSample is vulnerable to the attack. This makes it a slightly larger target compared to fH, but also with a much lower success probability. In total, a fault inside the vulnerable portions can be exploited with a probability of 56 %. These cover 33.5 % of execution time, thus approximately 19 % of random faults anywhere during signing lead to key recovery.

6 Countermeasures

When presenting new attacks, a discussion on potential countermeasures should never be missing. For this reason, we present the applicability and effectiveness of three generic countermeasures against the fault attacks described in this work. For each of these methods, we give the runtime costs and state which fault scenarios will be mitigated by it. A summary of the latter is shown in Table 6.

Double computation. While determinism leads to the applicability of differential fault attacks in the first place, it can also be used as a countermeasure against such attacks. Concretely, many faults can be detected by running the signature algorithm twice and testing the output for equality. This obviously doubles execution time. The countermeasure can be defeated by either injecting an identical fault twice, which can be challenging, or by using a permanent fault, e.g., in scenario fA_ρ with the seed ρ.

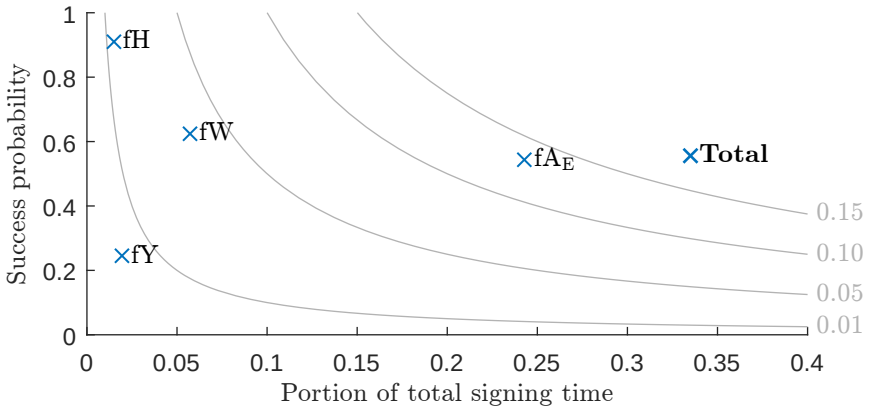

Figure 3: Comparison of scenarios regarding runtime as portion of total signing time (from Table 5) vs. success probability (from Table 4). Lines of constant product are drawn in solid gray.

Verification-after-sign. Many of the presented attack scenarios lead to signatures being invalid. Thus, performing signature verification after signing is an effective countermeasure. As runtime costs of verification are less than one-third of signing (see [LDK+17]), this option is also much more efficient than double computation. As a downside, however, it cannot detect faults injected into the sampling of $\mathbf{y}$ as this yields valid signatures. Permanent faults in ρ can be detected if $tr = \mathsf{CRH}(\rho||\mathbf{t}_1)$ is computed during key generation and then stored as part of the private key (as described in Algorithm 1). If tr is recomputed in the signing algorithm using the public key, then faults in ρ are not detectable.

Additional randomness. A final and very simple countermeasure is to re-randomize the deterministic sampling of the noise $\mathbf{y}$. One can simply sample a random $r \leftarrow 0,1^{256}$ and then invoke $\mathbf{y} := \mathsf{DeterministicSample}(K||\mu||\kappa||r)$. This effectively mitigates the differential fault attack as the faulted call to the signing algorithm uses different $\mathbf{y}$ and thus $\Delta\mathbf{y} \neq 0$. Still, the protection against nonce reuse (using the same $\mathbf{y}$ for different messages) is kept intact. For instance, using a constant r effectively reverts signing to its deterministic version. Apart from its simplicity, this countermeasure also has a negligible runtime overhead. Additionally, unlike straight-forward implementations of the two previous countermeasures, it is single-pass and so does not require to keep a copy of the message in memory. Note that this countermeasure was already proposed in the context of EdDSA [ABF+17], but as shown it can also be applied to lattice-based signatures.

There are, however, also considerable downsides of this countermeasure. First, unlike the two previous countermeasures, this countermeasure is probabilistic and requires some source of entropy, i.e., a true random number generator. Such a generator might not be available on all devices, especially low-resource ones. And second, this countermeasure violates the security proof of Dilithium. Kiltz, Lyubashevksy, and Schaffner [KLS17] present a tight proof in the quantum random oracle model (QROM) based on the hardness of MLWE, MSIS, and a new problem called SelfTargetMSIS. They require the signature scheme to be deterministic. They do give an alternative proof for a probabilistic version of Dilithium, yet it is not tight and loses security linearly in the number of observed unique signatures per message.

Thus, introducing this countermeasure voids provable security guarantees, albeit no concrete attack is known. The Dilithium authors *"still recommend using deterministic*

Table 6: Applicable countermeasures

	fA_ρ	fA_E	fY	fW	fH
Double computation	✗	✓	✓	✓	✓
Verification-after-sign	✓/✗*	✓	✗	✓	✓
Additional randomness†	✓	✓	✓	✓	✓

* Under certain conditions.

† Not supported by proof of Dilithium [KLS17].

signatures except in environments that may be vulnerable to the aforementioned side-channel attacks" [LDK+17]. However, determining whether or not an environment is vulnerable is not easy, as clearly shown by the Rowhammer bug.

References

[ABB+16] Sedat Akleylek, Nina Bindel, Johannes A. Buchmann, Juliane Krämer, and Giorgia Azzurra Marson. An efficient lattice-based signature scheme with provably secure instantiation. In *AFRICACRYPT*, volume 9646 of *LNCS*, pages 44–60. Springer, 2016.

[ABB+17] Erdem Alkim, Nina Bindel, Johannes A. Buchmann, Özgür Dagdelen, Edward Eaton, Gus Gutoski, Juliane Krämer, and Filip Pawlega. Revisiting TESLA in the quantum random oracle model. In *PQCrypto*, volume 10346 of *LNCS*, pages 143–162. Springer, 2017. Full version available at https://ia.cr/2015/755.

[ABF+17] Christopher Ambrose, Joppe W. Bos, Björn Fay, Marc Joye, Manfred Lochter, and Bruce Murray. Differential attacks on deterministic signatures. Cryptology ePrint Archive, Report 2017/975, 2017. To appear at CT-RSA 2018.

[AFG13] Martin R. Albrecht, Robert Fitzpatrick, and Florian Göpfert. On the efficacy of solving LWE by reduction to unique-SVP. In *ICISC*, volume 8565 of *LNCS*, pages 293–310. Springer, 2013.

[BAA+17] Nina Bindel, Sedat Akleylek, Erdem Alkim, Paulo S. L. M. Barreto, Johannes Buchmann, Edward Eaton, Gus Gutoski, Juliane Krämer, Patrick Longa, Harun Polat, Jefferson E. Ricardini, and Gustavo Zanon. qTESLA. Submission to the NIST Post-Quantum Cryptography Standardization [NIS], 2017. https://tesla.informatik.tu-darmstadt.de/de/tesla/.

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

[BCN+06] Hagai Bar-El, Hamid Choukri, David Naccache, Michael Tunstall, and Claire Whelan. The sorcerer's apprentice guide to fault attacks. *Proceedings of the IEEE*, 94(2):370–382, 2006.

[BDL+11] Daniel J. Bernstein, Niels Duif, Tanja Lange, Peter Schwabe, and Bo-Yin Yang. High-speed high-security signatures. In *CHES*, volume 6917 of *LNCS*, pages 124–142. Springer, 2011.

[BDPV07] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. Sponge functions. ECRYPT Hash Workshop, 2007.

[bms10] "bushing", "marcan", and "sven". PS3 epic fail. 27th Chaos Communication Congress, 2010. https://events.ccc.de/congress/2010/Fahrplan/events/4087.en.html.

[BP16] Alessandro Barenghi and Gerardo Pelosi. A note on fault attacks against deterministic signature schemes. In *IWSEC*, volume 9836 of *LNCS*, pages 182–192. Springer, 2016.

[DDLL13] Léo Ducas, Alain Durmus, Tancrède Lepoint, and Vadim Lyubashevsky. Lattice signatures and bimodal gaussians. In *CRYPTO (1)*, volume 8042 of *LNCS*, pages 40–56. Springer, 2013.

[DLL+17] Léo Ducas, Tancrède Lepoint, Vadim Lyubashevsky, Peter Schwabe, Gregor Seiler, and Damien Stehlé. CRYSTALS – Dilithium: Digital signatures from module lattices. Cryptology ePrint Archive, Report 2017/633, 2017. Publication to [LDK+17], also appeared at CHES 2018.

[EFGT16] Thomas Espitau, Pierre-Alain Fouque, Benoît Gérard, and Mehdi Tibouchi. Loop-abort faults on lattice-based Fiat-Shamir and hash-and-sign signatures. In *SAC*, volume 10532 of *LNCS*, pages 140–158. Springer, 2016.

[EFGT17] Thomas Espitau, Pierre-Alain Fouque, Benoît Gérard, and Mehdi Tibouchi. Side-channel attacks on BLISS lattice-based signatures: Exploiting branch tracing against strongSwan and electromagnetic emanations in microcontrollers. In *CCS*, pages 1857–1874. ACM, 2017.

[FS86] Amos Fiat and Adi Shamir. How to prove yourself: Practical solutions to identification and signature problems. In *CRYPTO*, volume 263 of *LNCS*, pages 186–194. Springer, 1986.

[GBHLY16] Leon Groot Bruinderink, Andreas Hülsing, Tanja Lange, and Yuval Yarom. Flush, Gauss, and Reload - A cache attack on the BLISS lattice-based signature scheme. In *CHES*, volume 9813 of *LNCS*, pages 323–345. Springer, 2016.

[GLP12] Tim Güneysu, Vadim Lyubashevsky, and Thomas Pöppelmann. Practical lattice-based cryptography: A signature scheme for embedded systems. In *CHES*, volume 7428 of *LNCS*, pages 530–547. Springer, 2012.

[GMM16] Daniel Gruss, Clémentine Maurice, and Stefan Mangard. Rowhammer.js: A remote software-induced fault attack in JavaScript. In *DIMVA*, volume 9721 of *LNCS*, pages 300–321. Springer, 2016.

[HLS18] Andreas Hülsing, Tanja Lange, and Kit Smeets. Rounded Gaussians. In *PKC*, volume 10770 of *LNCS*, pages 728–757. Springer, 2018.

[HPS+14] Jeffrey Hoffstein, Jill Pipher, John M. Schanck, Joseph H. Silverman, and William Whyte. Practical signatures from the partial Fourier recovery problem. In *ACNS*, volume 8479 of *LNCS*, pages 476–493. Springer, 2014.

[KDK+14] Yoongu Kim, Ross Daly, Jeremie Kim, Chris Fallin, Ji-Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors. In *ISCA*, pages 361–372. IEEE Computer Society, 2014.

[Kel18] Julian Kelly. A preview of Bristlecone, Google's new quantum processor, 2018. https://research.googleblog.com/2018/03/a-preview-of-bristlecone-googles-new.html.

[KLS17] Eike Kiltz, Vadim Lyubashevsky, and Christian Schaffner. A concrete treatment of Fiat-Shamir signatures in the quantum random-oracle model. Cryptology ePrint Archive, Report 2017/916, 2017. To appear at EUROCRYPT 2018.

[KRR+18] Angshuman Karmakar, Sujoy Sinha Roy, Oscar Reparaz, Frederik Vercauteren, and Ingrid Verbauwhede. Constant-time discrete Gaussian sampling. unpublished, 2018. https://www.esat.kuleuven.be/cosic/publications/article-2822.pdf.

[LDK+17] Vadim Lyubashevsky, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Peter Schwabe, Gregor Seiler, and Damien Stehlé. CRYSTALS-Dilithium. Submission to the NIST Post-Quantum Cryptography Standardization [NIS], 2017. https://pq-crystals.org/dilithium.

[Lyu09] Vadim Lyubashevsky. Fiat-Shamir with Aborts: Applications to lattice and factoring-based signatures. In *ASIACRYPT*, volume 5912 of *LNCS*, pages 598–616. Springer, 2009.

[Moo17] Dustin Moody. The ship has sailed - the NIST post-quantum crypto "competition". Invited Talk at ASIACRYPT 2017, 2017. https://csrc.nist.gov/CSRC/media//Projects/Post-Quantum-Cryptography/documents/asiacrypt-2017-moody-pqc.pdf.

[MOP07] Stefan Mangard, Elisabeth Oswald, and Thomas Popp. *Power analysis attacks - revealing the secrets of smart cards*. Springer, 2007.

[MW17] Daniele Micciancio and Michael Walter. Gaussian sampling over the integers: Efficient, generic, constant-time. In *CRYPTO (2)*, volume 10402 of *LNCS*, pages 455–485. Springer, 2017.

[NIS] NIST. Post-quantum cryptography standardization. https://csrc.nist.gov/projects/post-quantum-cryptography/post-quantum-cryptography-standardization.

[OSPG16] Tobias Oder, Tobias Schneider, Thomas Pöppelmann, and Tim Güneysu. Practical CCA2-secure and masked ring-lwe implementation. Cryptology ePrint Archive, Report 2016/1109, 2016. To appear at CHES 2018.

[PBY17] Peter Pessl, Leon Groot Bruinderink, and Yuval Yarom. To BLISS-B or not to be: Attacking strongSwan's implementation of post-quantum signatures. In *CCS*, pages 1843–1855. ACM, 2017.

[Pes16] Peter Pessl. Analyzing the shuffling side-channel countermeasure for lattice-based signatures. In *INDOCRYPT*, volume 10095 of *LNCS*, pages 153–170, 2016.

[Por13] T. Pornin. Deterministic usage of the digital signature algorithm (DSA) and elliptic curve digital signature algorithm (ECDSA). RFC 6979, 2013. https://tools.ietf.org/html/rfc6979.

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

[PSS+17] Damian Poddebniak, Juraj Somorovsky, Sebastian Schinzel, Manfred Lochter, and Paul Rösler. Attacking deterministic signature schemes using fault attacks. Cryptology ePrint Archive, Report 2017/1014, 2017.

[RRVV15] Oscar Reparaz, Sujoy Sinha Roy, Frederik Vercauteren, and Ingrid Verbauwhede. A masked ring-lwe implementation. In *CHES*, volume 9293 of *LNCS*, pages 683–702. Springer, 2015.

[Saa17] Markku-Juhani O. Saarinen. Arithmetic coding and blinding countermeasures for lattice signatures. *Journal of Cryptographic Engineering*, 2017.

A Dilithium Parameter Sets

Table 7 shows all parameter sets specified by the Dilithium authors [LDK+17]. Parameters $(n, q, \text{weight}(c), \gamma_1, \gamma_2)$ are identical across all sets and thus allow for simpler implementation and switching of security levels.

Table 7: Dilithium Parameter Sets

	I weak	II medium	III recommended	IV high
n	256	256	256	256
q	8380417	8380417	8380417	8380417
d	14	14	14	14
weight(c)	60	60	60	60
γ_1	523776	523776	523776	523776
γ_2	261888	261888	261888	261888
(k, ℓ)	(3, 2)	(4, 3)	(5, 4)	(6, 5)
η	7	6	5	3
β	375	325	275	175
ω	64	80	96	120

B Description of qTESLA

In this section we briefly describe the signature scheme qTESLA, for more details we refer to the original submission [BAA+17]. In Algorithms 7, 8, and 9, we give slightly simplified versions of key generation, signing, and verification, respectively. Note its similarity to Dilithium, we highlight this similarity by stating the corresponding variable and function names in Table 8. The main difference between Dilithium and qTESLA is that the latter is based on Ring-LWE and thus operates on polynomials in $\mathbb{Z}_q[x]/(x^n + 1)$ with $n \geq 1024$. Dilithium is based on the Module-LWE assumption and uses vectors/matrices of polynomials in a fixed base ring $\mathbb{Z}_q[x]/(x^{256} + 1)$.

All attacks for Dilithium described in Section 3 can easily be adapted to qTESLA, with obviously differing success probabilities. Note that in qTESLA the public key t is not compressed, thus recovering e (which corresponds to $\mathbf{s}_2$ in Dilithium) is trivial as soon as s (corresponds to $\mathbf{s}_1$) is known. No adapted signature algorithm (as described in Section 4) is needed.

Algorithm 7 qTESLA Key Generation

Output: Keypair (pk, sk)
1: $\text{seed}_a \leftarrow \{0,1\}^{256}, \text{seed}_y \leftarrow \{0,1\}^{256}$
2: $a \in \mathcal{R}_q \coloneqq \text{GenA}(\text{seed}_a)$
3: **do**
4: $\quad s \in \mathcal{R}_q \leftarrow D_\sigma, e \in \mathcal{R}_q \leftarrow D_\sigma$ ▷ Discrete Gaussian distribution D_σ
5: **while** s and e do not fulfill certain criteria
6: $t \coloneqq as + e \bmod q$
7: **return** $(pk = (\text{seed}_a, t), sk = (s, e, \text{seed}_y, \text{seed}_a))$

Algorithm 8 qTESLA Sign (simplified)

Input: Message M, private key $sk = (s, e, \text{seed}_y, \text{seed}_a)$
Output: Signature $\sigma = (c, z)$
1: $a \in \mathcal{R}_q \coloneqq \text{GenA}(\text{seed}_a)$
2: counter $\coloneqq 0$
3: rand $\coloneqq \mathsf{PRF}_1(\text{seed}_y, M)$
4: **do**
5: $\quad$ y $\coloneqq \mathsf{PRF}_2(\text{rand}, \text{counter})$
6: $\quad v \coloneqq ay \bmod q$
7: $\quad c \coloneqq \mathsf{H}(\text{Round}(v), M)$
8: $\quad z \coloneqq y + sc$
9: $\quad$ counter $\coloneqq$ counter $+ 1$
10: **while** $\text{Reject}(z, v, c, sk)$
11: **return** $\sigma = (c, z)$

Algorithm 9 qTESLA Verify (simplified)

Input: Public key $pk = (\text{seed}_a, t)$, message M, signature $\sigma = (c, z))$
1: $a \in \mathcal{R}_q \coloneqq \mathsf{GenA}(\text{seed}_a)$
2: $w \coloneqq az - tc \bmod q$
3: **return** $c = \mathsf{H}(\text{Round}(w), M)$

Table 8: Comparison of variable/parameter names and function names for Dilithium and qTESLA. Only differing names are listed.

Variables:

Dilithium	ρ	K	$\mathbf{s}_1$	$\mathbf{s}_2$	κ	μ	$\mathbf{w}$
qTESLA	seed_a	seed_y	s	e	counter	rand	v

Functions:

Dilithium	ExpandA	CRH	DeterministicSample	HighBits
qTESLA	GenA	PRF_1	PRF_2	Round