IACR Transactions on Cryptographic Hardware and Embedded Systems
ISSN 2569-2925, Vol. 2023, No. 4, pp. 367–392. DOI:10.46586/tches.v2023.i4.367-392

Loop Aborts Strike Back: Defeating Fault Countermeasures in Lattice Signatures with ILP

Vincent Quentin Ulitzsch[1], Soundes Marzougui[1,2], Alexis Bagia[1], Mehdi Tibouchi[3] and Jean-Pierre Seifert[1,4]

[1] Technical University Berlin, Berlin, Germany
vincent@sect.tu-berlin.de, soundes.marzougui@tu-berlin.de, a.bagia@campus.tu-berlin.de, jean-pierre.seifert@tu-berlin.de
[2] STMicroelectronics, Diegem, Belgium
[3] NTT Social Informatics Laboratories, Tokyo, Japan
mehdi.tibouchi@ntt.com
[4] Fraunhofer Institute for Secure Information Technology, Darmstadt, Germany

Abstract. At SAC 2016, Espitau et al. presented a loop-abort fault attack against lattice-based signature schemes following the Fiat–Shamir with aborts paradigm. Their attack recovered the signing key by injecting faults in the sampling of the commitment vector (also called masking vector) $\mathbf{y}$, leaving its coefficients at their initial zero value. As possible countermeasures, they proposed to carry out the sampling of the coefficients of $\mathbf{y}$ in shuffled order, or to ensure that the masking polynomials in $\mathbf{y}$ are not of low degree. In this paper, we show that both of these countermeasures are insufficient. We demonstrate a new loop-abort fault injection attack against Fiat–Shamir with aborts lattice-based signatures that can recover the secret key from faulty signatures even when the proposed countermeasures are implemented. The key idea of our attack is that faulted signatures give rise to a noisy linear system of equations, which can be solved using integer linear programming. We present an integer linear program that recovers the secret key efficiently in practice, and validate the efficacy of our attack by conducting a practical end-to-end attack against a shuffled version of the Dilithium reference implementation, mounted on an ARM Cortex M4. We achieve a full (equivalent) key recovery in under 3 minutes total execution time (including signature generation), using only 5 faulted signatures. In addition, we conduct extensive theoretical simulations of the attack against Dilithium. We find that our method can achieve key recovery in under 5 minutes given a (sufficiently large) set of signatures where just *one* of the coefficients of $\mathbf{y}$ is zeroed out (or left at its initial value of zero). Furthermore, we find that our attack works against all security levels of Dilithium. Our attack shows that protecting Fiat–Shamir with aborts lattice-based signatures against fault injection attacks cannot be achieved using the simple countermeasures proposed by Espitau et al. and likely requires significantly more expensive countermeasures.

Keywords: Fault analysis · Fiat–Shamir with aborts · Dilithium · Side-channel attacks · Integer linear programming

1 Introduction

The potential advent of large-scale quantum computers in the near future threatens to undermine the security of currently deployed public-key cryptography [Mos18], as Shor's algorithm would render public-key cryptography based on the hardness of integer factoring or discrete logarithms insecure.

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

Recent advances in quantum computing [Int19, DCG21, Rig21, AAB+19, Mos18, Gri19] point to the urgency of designing and implementing quantum-resistant cryptographic schemes that are suitable for real-world deployment, whether in terms of security, performance or practicality of implementation. This is being addressed by the ongoing NIST-led process to evaluate and standardize post-quantum primitives for public-key encryption, key encapsulation, and signatures [Cen20]. Schemes submitted to the NIST call for proposals have been subject to heavy scrutiny from the research community, and various proposed candidates were eliminated because they were found to be insecure. In practice, however, the extensive study of the theoretical security of proposed schemes alone is insufficient. In order to maintain the intended security guarantees of quantum-secure algorithms, it is crucial to ensure proper implementation and device behavior during their execution, as any shortcomings in these areas can leave the cryptographic schemes vulnerable to side-channel attacks. A case in point is the security of implementations against fault-injection attacks, which pose a significant security threat to schemes meant for real-world deployment. The foreseeable use cases for post-quantum secure signature schemes as well as key encapsulation mechanisms include signing banking transactions with a credit card or a hardware wallet and providing identification in IoT devices or vehicular networks. These use cases entail executing the cryptographic schemes on microcontrollers and smart cards, which are particularly susceptible to fault attacks.

Fault attacks induce an error during the execution of a cryptographic algorithm which can lead to unexpected behavior. Based on subsequent mathematical analysis of the faulty output, e.g., the signature, the adversary is able to recover the secret key. Hence, the security of the system is compromised. For post-quantum secure signature schemes, fault attacks have successfully been used to recover a secret key [RJH+19, AKKM22, Del21, GKPM18].

At SAC 2016, Espitau et al. presented a powerful type of fault-attacks against lattice-based Fiat–Shamir with aborts signature schemes [EFGT16]. Fiat–Shamir with aborts signatures rely on a commitment value $\mathbf{y}$, sampled during signature generation, to hide information about the secret key. This commitment value is a polynomial or a vector of polynomials, whose coefficients are sampled sequentially in a loop. Using a fault to abort the sampling loop early on causes the commitment polynomial(s) to be of low degree, as most of its coefficients will be set to zero (in some implementations, this could also be a different fixed, known value, and that case easily reduces to the case of low degree). As a result, the faulty signature leaks information, allowing the secret key to be recovered efficiently via (low dimensional) lattice reduction algorithms. The same type of attack can also be mounted through "zeroing faults", where the attacker sets a BRAM memory cell to its initial state, causing the commitment polynomial to be altered and the corresponding signature to leak information.

To mitigate loop-abort faults from enabling an attacker to recover the secret key, Espitau et al. suggest two cryptographic countermeasures: first, to shuffle the order in which the coefficients are generated (which ensures, with good probability, that generated polynomials are no longer of low degree, thwarting the attack); second, to check after sampling that the generated polynomials are of high degree (preventing the lattice reduction technique directly) and rejecting otherwise.

Contributions In this work, we show that neither the shuffling countermeasure nor the rejection of low-degree polynomials suffices to prevent loop abort or zeroing fault attacks. Our key idea is to formulate integer linear programs (ILPs) that can efficiently recover the secret signing key from a limited number of faulty signatures even in the presence of the aforementioned countermeasures presented in [EFGT16]. We verify that this integer linear program is efficiently solvable in practice (heuristically, because its rational relaxation often already provides a near-optimal solution), and thus allows us to circumvent the countermeasures easily.

We validate the effectiveness of our attack against Dilithium, one of the post-quantum signature schemes that NIST has selected for standardization, and an example of a Fiat–Shamir with aborts lattice-based signature scheme. We show — through extensive simulations — that as little as *one single skipped coefficient* is sufficient to recover Dilithium's secret key in under 5 minutes, given sufficiently many faulty signatures. Moreover, we demonstrate that our attack works against all security levels of Dilithium (and it would apply similarly to any other Fiat–Shamir with aborts lattice-based scheme, like BLISS or qTESLA).

We also present an end-to-end proof of concept attack against a shuffled Dilithium implementation mounted on an ARM Cortex M4, recovering the key using only 5 faulted signatures. The total execution time of our attack, including both the time for signature generation as well as the time required for key recovery, is under 4 minutes.

The rest of the paper is structured as follows. After reviewing related work in Section 2 and giving the necessary background (Section 3), we explain the attack presented by Espitau et al. in Section 4. We describe the high-level idea and the ILP formulation in Section 5 and show concrete attacks against Dilithium in Section 5.3. We present theoretical simulations of the attack in Section 6 and discuss the practical end-to-end attack in Section 7.

We also examine the required countermeasures to avoid our attack in Section 8. We conclude the paper and explore its implications in Section 9.

2 Related Work

Loop-abort fault attacks were introduced by Page and Vercauteren [PV06] to subvert pairing-based cryptography and have also been used to attack pairing-based cryptography in practice [BDSG+14]. They have since been successfully utilized to attack post-quantum cryptography schemes as well. Gelin and Wesolowski demonstrate that one can attack supersingular isogeny cryptosystems by exploiting the iterative structure of the secret isogeny computation. The attacker can mount a loop-abort attack on the loop that computes a party's isogeny, and use the resulting information leak to recover the victim's secret key [GW17]. Additionally, the multivariate family of post-quantum cryptographic schemes has also been subjected to loop-abort attacks. Hashimoto et al. [HTS11] have presented general fault attacks on multivariate schemes. In these attacks, by deliberately aborting the random sampling process of sensitive vectors (called vinegar values), the authors were able to partially recover the secret key. Consequently, Aulbach et al. further demonstrated the practicality of this attack by applying it to the multivariate scheme Rainbow [AKKM22].

Multiple fault injection attacks against unprotected implementations of lattice-based cryptography have been demonstrated. For instance, McCarthy et al. introduced loop abort attacks on FALCON, utilizing the aborting recursion or zeroing (BEARZ) technique to target the FALCON sampler. This attack exploited faults within the FALCON signature scheme to gain access to private key information [MHS+19]. Furthermore, Dilithium and qTESLA are also vulnerable to differential fault attacks [BP18]. In those attacks, the attacker causes the signer to sign the same message twice and injects a fault to perturb the generation of the challenge vector in one of the two executions, whereas the other execution completes correctly. This causes a nonce reuse scenario, in which the same commitment vector $\mathbf{y}$ is used for two different challenge vectors $c \neq c'$. Doing so allows an attacker to recover the secret key, given a faulted and a corresponding non-faulted signature. Being able to cause a device to sign the *same* message twice is a strong attacker model; in contrast, our attack does not require any control over the signed messages.

In [BBK16], Bindel et al. provided an extensive overview of fault attacks against lattice-based signature schemes. One of the attacks considered in the paper zeroes out

a consecutive block of coefficients of the commitment polynomial $\mathbf{y}$ during signature generation. To recover the secret key, the attack presented in [BBK16] relies on the attacker's capability to determine which coefficients were zeroed out through statistical observations; the paper argues that an attacker can unambiguously determine which of $\mathbf{y}$'s coefficients were zeroed by observing the corresponding signature output $\mathbf{z}$. This is a consequence of the distribution of the coefficient of $\mathbf{z}$; in qTESLA (the scheme considered in that paper), the expected value of a signature coefficient $E[z_j]$ varies significantly if the corresponding commitment vector coefficient is zero. Thus, the position of the zeroed coefficients in $\mathbf{y}$ could be easily determined by simply observing the corresponding signature coefficients $\mathbf{z}$. This attack does not work, however, once the coefficients of $\mathbf{y}$ are shuffled because the zeroed coefficients do not appear consecutively anymore. As a result, it is infeasible to determine, with perfect accuracy, whether a coefficient y_j has been zeroed out solely by observing the corresponding signature output coefficient z_j.

A potential power side-channel attack against a shuffled BLISS implementation is presented in [Pes16]. The authors of [Pes16] show that it is possible to *unshuffle* the shuffled coefficients by leveraging statistical observations. The unshuffling relies on a list of *all* sampled coefficients, obtained through power side-channel analysis. This information is not present in the case of a loop-abort fault attack against shuffled sampling. Instead, the attacker only knows that some coefficients are zero, but does not know anything about the remaining coefficients.

3 Background

For any integer q, the ring $\mathbb{Z}_q$ is represented by the set $[-q/2, q/2) \cap \mathbb{Z}$. For an even positive integer α, we define $r' = r \bmod^{\pm} \alpha$ to be the unique element r' in the range $-\frac{\alpha}{2} < r' \leq \frac{\alpha}{2}$, such that $r' \equiv r \mod \alpha$. Analogously, for an odd positive integer α, we define $r' = r \bmod^{\pm} \alpha$ as $-\frac{\alpha-1}{2} \leq r' \leq \frac{\alpha-1}{2}$, such that $r' \equiv r \mod \alpha$. For any positive integer α, we denote by $r' = r \bmod^{+} \alpha$ the unique element r' in the range $0 \leq r' < \alpha$ such that $r' \equiv r \mod \alpha$. We denote $\mathbb{Z}_q[X] = (\mathbb{Z}/q\mathbb{Z}[X])$ as the set of polynomials with integer coefficients modulo q. We define $R_q = \mathbb{Z}_q[X]/(X^n + 1)$ as the ring of polynomials with integer coefficients modulo q, reduced by the cyclotomic polynomial $X^n + 1$. For any $\alpha < q$, R_α refers to the set of polynomials with coefficients in $[-\alpha/2, +\alpha/2]$. For any integer $w \in \mathbb{Z}_q$, we define $||w||_\infty = |w \bmod^{\pm} q|$, and for a ring element $w \in R$ with $w = \sum_{i=0}^{n-1} w_i X^i$, we define $||w||_\infty = \max_i ||w_i||_\infty$. By $\mathtt{vec}(v)$ we denote the function which maps a ring element $v \in R_q$ to the vector obtained by concatenating all of the polynomial's coefficients.

Bold lowercase letters represent vectors with coefficients in R or R_q. Bold uppercase letters are matrices. Vector coefficients and elements in R or R_q are represented by Roman lowercase letters.

3.1 Lattices

A *lattice* Λ is a discrete subgroup of $\mathbb{R}^n$. Given $m \leq n$ linearly independent vectors $\mathbf{b_1}, ..., \mathbf{b_m} \in \mathbb{R}^n$, the lattice $\Lambda(\mathbf{b}_1, ..., \mathbf{b}_m)$ is the set of all integer linear combinations of the $\mathbf{b}_i$'s, i.e.,

$$\Lambda(\mathbf{b}_1, ..., \mathbf{b}_m) = \left\{ \sum_{i=1}^{m} x_i \mathbf{b}_i \;\middle|\; x_i \in \mathbb{Z} \right\},$$

where $\mathbf{b}_1, ..., \mathbf{b}_m$ form a *basis* of Λ and m is the *rank*. This paper considers full-rank lattices, i.e., with $m = n$. An *integer lattice* is a lattice for which the basis vectors are in

$\mathbb{Z}^n$. Usually, we consider elements modulo q, i.e., the basis vectors and coefficients are taken from $\mathbb{Z}_q$.

3.2 Fiat–Shamir With Aborts Scheme

Fiat and Shamir [FS87] introduced a technique to derive a digital signature scheme from an interactive (potentially zero-knowledge) identification scheme. This technique is canonically known as the Fiat–Shamir heuristic. In [Lyu09], Lyubashevsky builds on top of this technique to construct a signature scheme based on lattices. The technique is coined Fiat–Shamir with aborts. Fiat–Shamir with aborts signature schemes, as originally described by [Lyu09], base their security on the worst-case hardness of approximating the shortest vector problem within a factor of $O(n^2)$ in lattices corresponding to ideals in the ring $R = \mathbb{Z}[X]/(X^n + 1)$.

Figure 1 describes signing and verification in the Fiat–Shamir with aborts signature scheme. The scheme requires a family of collision-resistant homomorphic hash functions $\mathcal{H}$, mapping from D^ℓ to R, where $D \subseteq R$ is a restricted domain of R and ℓ is a positive integer. Additionally, it makes use of a random oracle $H : \{0,1\}^* \to D_c$, where $D_c \subseteq$ is a restricted domain of R, corresponding to polynomials with small coefficients.

In summary, the secret key $\mathbf{s}$ is a vector of ℓ polynomials in ring R with small coefficients. The public key consists of a hash function h drawn from $\mathcal{H}$ and the hash of the secret key $S = h(\mathbf{s})$. To sign a message μ, the signer samples a commitment vector of ℓ polynomials $\mathbf{y} \in D_y^\ell$ (sometimes also referred to as a masking vector), where $D_y \subseteq R$ with restricted coefficients, and then hashes $h(\mathbf{y})$ and the message μ to a challenge polynomial $c \in R$ with small coefficients, through the random oracle function, i.e., $c = H(h(\mathbf{y}), \mu)$. The signature is the tuple $(\mathbf{z} = \mathbf{s}c + \mathbf{y}, c)$. The process is repeated in case the signature's coefficients do not fall in a certain range. To verify a signature $\mathbf{z}$, the verifier ensures that $\mathbf{z}$'s coefficients fall in the correct range and that $c = H(h(\mathbf{z}) - Sc, \mu)$. It holds that $h(\mathbf{z}) - Sc = h(\mathbf{z}) - h(\mathbf{s})c$ and since h is homomorphic, it holds that $h(\mathbf{z}) - h(\mathbf{s}c) = h(\mathbf{z} - \mathbf{s}c) = h(\mathbf{y})$. Thus, an honest signer can convince the verifier.

3.3 The BLISS Signature Scheme

BLISS can be seen as a ring-based optimization of the earlier lattice-based scheme of Lyubashevsky [Lyu09], sharing the same "Fiat–Shamir with aborts" structure [Lyu09]. A simplified description of the signing process is given in Algorithm 1. The public key is an NTRU-like ratio of the form $\mathbf{a}_q = \mathbf{s}_2/\mathbf{s}_1 \mod q$, and the secret key is composed of the two small and sparse polynomials $\mathbf{s}_1, \mathbf{s}_2 \in R$, where $R = \mathbb{Z}[X]/(X^n + 1)$.

To sign a message $\mu \in \{0,1\}^*$, we first generate commitment vectors $\mathbf{y}_1, \mathbf{y}_2 \in R$ with normally distributed coefficients. Then, we compute a hash c of the message μ together with $\mathbf{u} = -\mathbf{a}_q\mathbf{y}_1 + \mathbf{y}_2 \mod q$ using a cryptographic hash function H modeled as a random oracle taking values in the set of elements of R. The signature is the triple $(c, \mathbf{z}_1, \mathbf{z}_2)$, where $\mathbf{z}_i = \mathbf{y}_i + (-1)^b\mathbf{s}_i c \mod 2q$ for some random bit b. A rejection condition ensures the independence of $\mathbf{z}$ and $\mathbf{s}$.

3.4 The Dilithium Signature Scheme

The Dilithium signature scheme is based on the Fiat–Shamir with aborts structure [Lyu09]. Its security is tied to the hardness of the Modular-Learning with Errors and Modular-Short Integer Solution problems. Accordingly, its operations are carried out over the ring $R_q = \mathbb{Z}_q[X]/(X^n + 1)$. Following the notation of the Dilithium specification [BDK+21], we will use the following notation. We denote by B_τ the ball with the set of elements of R that have τ coefficients that are either -1 or 1 and the rest are 0. Let S_η denote the set of all elements $w \in R$ such that $\|w\|_\infty \leq \eta$. By $\tilde{S}_\eta$ we denote the set $\{w \bmod^{\pm} 2\eta : w \in R\}$.

Signing Key : $\mathbf{s} \xleftarrow{\$} D_s^\ell$

Verification Key : $h \xleftarrow{\$} \mathcal{H}(R, D, \ell), S \leftarrow h(\mathbf{s})$

Random Oracle : $H : \{0,1\}^* \to D_c$

Sign	Verify
1 : $\mathbf{y} \xleftarrow{\$} D_y^\ell$	1 : Accept iff $\mathbf{z} \in G^\ell$ and $c = H(h(\mathbf{z}) - Sc, \mu)$
2 : $c \leftarrow H(h(\mathbf{y}), \mu)$	
3 : $\mathbf{z} \leftarrow \mathbf{s}c + \mathbf{y}$	
4 : if $\mathbf{z} \notin G^\ell$, then goto step 1	
5 : **return** $(\mathbf{z}, c)$	

Figure 1: Fiat–Shamir with aborts Signature Scheme. $G \subseteq R$ consists of polynomials with restricted coefficients. We denote by $x \xleftarrow{\$} S$ that x is assigned an element of a set S, sampled uniformly at random.

Algorithm 1 BLISS signature generation

Input: A message μ, public key $\mathbf{a}_1$, secret key $\mathbf{S} = (\mathbf{s}_1, \mathbf{s}_2)$

Output: A signature $(\mathbf{z}_1, \mathbf{z}_2^\dagger, c) \in \mathbb{Z}_{2q}^n \times \mathbb{Z}_p^n \times \{0,1\}^n$ of the message μ

1: $\mathbf{y}_1, \mathbf{y}_2 \leftarrow D_{\mathbb{Z}^n,\sigma}$ ▷ $D_{\mathbb{Z}^n,\sigma}$ is the discrete Gaussian distribution over $\mathbb{Z}^n$ with standard deviation σ.
2: $\mathbf{u} = \zeta \mathbf{a}_1 \mathbf{y}_1 + \mathbf{y}_2 \mod 2q$ ▷ Where ζ is defined such that $\zeta \cdot (q-2) = 1 \mod 2q$.
3: $c = H(\lfloor \mathbf{u} \rceil_d \mod p, m)$
4: choose a uniform random bit b
5: $\mathbf{z}_1 = \mathbf{y}_1 + (-1)^b \mathbf{s}_1 c \mod 2q$
6: $\mathbf{z}_2 = \mathbf{y}_2 + (-1)^b \mathbf{s}_2 c \mod 2q$
7: rejection sampling: **restart** to step 2 except with probability based on σ, $\|\mathbf{S}c\|$, $\langle \mathbf{S}c \rangle$
8: $\mathbf{z}_2^\dagger = (\lfloor \mathbf{u} \rceil_d - \lfloor \mathbf{u} - \mathbf{z}_2 \rceil_d) \mod p$
9: **return** $(\mathbf{z}_1, \mathbf{z}_2^\dagger, c)$

In summary, the public key consists of a matrix $\mathbf{A} \in R_q^{k \times \ell}$, generated uniformly at random and a vector $\mathbf{t} = \mathbf{A}\mathbf{s}_1 + \mathbf{s}_2$. The secret key vector $\mathbf{s}_1 \in R_q^\ell$ and $\mathbf{s}_2 \in R_q^k$ are vectors of small polynomials, with each coefficient being chosen uniformly at random from a small range $\{-\eta, \ldots, \eta\}$.

To sign a message μ, the following steps are taken: A commitment vector $\mathbf{y} \in R_q^\ell$ is first generated by sampling $\ell \cdot n$ uniformly random coefficients from the range $\{-\gamma_1 + 1, \ldots, \gamma_1\}$. Next, the message μ together with $\mathbf{A}\mathbf{y}$ is hashed to obtain a sparse polynomial $c = H(\mu, \mathbf{A}\mathbf{y})$, a ring element with τ coefficient being -1 or 1, and the rest being zero. Finally, if a certain rejection condition is met (mostly stating that $\mathbf{z}$ has its coefficients in a small interval) then the signature tuple $\mathbf{z} = ((c\mathbf{s} + \mathbf{y}), c)$ is outputted. Otherwise, the signature generation is repeated.

A more detailed description of Dilithium's key generation and signing procedure is given in Algorithm 2 and Algorithm 3. The key generation includes the sampling of additional randomness seeds ρ, ρ', K to be used in the deterministic signing procedure later on. Dilithium also supports a non-deterministic version, where it samples fresh randomness for each signature. The key generation and signing procedures additionally make use of the following functions. An extensible output function H, mapping to the appropriate domain. The functions `ExpandA` and `ExpandS` map a uniform seed $\{0,1\}^{256}$ to a matrix $\mathbf{A} \in R_q^{k \times \ell}$ and two vectors $(\mathbf{s}_1, \mathbf{s}_2) \in S_\eta^\ell \times S_\eta^k$, respectively. The function `SampleInBall` takes as input a seed ρ and returns a random 256-element array with τ elements being ± 1 and the rest 0. To shrink the output size, the functions $\texttt{HighBits}_q(r, \alpha)$ describes a function

that returns an r_1 such that $r = r_1 \cdot \alpha + r_0 \mod q$, whereas $\texttt{LowBits}_q(r, \alpha)$ returns r_0. The $\texttt{MakeHint}_q(z, r, \alpha)$ procedure produces hints to help guess the shrunk bits of a sum. The function $\texttt{Power2Round}_q(r, d)$ returns the tuple $((r' - r_0)/2^d, r_0)$, where $r' = r \mod^+ q$ and $r_0 = r' \mod^\pm 2^d$. Lastly, the `ExpandMask` function maps a seed ρ' and a nonce κ to $\tilde{S}^{\ell}_{\gamma_1}$, leveraging an Extensible Output Function (XOF) in the process. The function first seeds the XOF with ρ' and κ and then converts the resulting byte buffer to a vector of polynomials $\tilde{S}^{l}_{\gamma_1}$. The parameters $k, \ell, \gamma_1, \gamma_2, \omega, \tau, \beta$ are dependent on Dilithium's security level.

To verify a signature, the verifier first computes $\mathbf{w}'_1$ as the high-order bits of $\mathbf{Az} - c\mathbf{t}$, and then accepts if all the coefficients of $\mathbf{z}$ are less than $\gamma_1 - \beta$ and if c matches the hash of the signed message concatenated with $\mathbf{w}_1$.

We stress that, although the private key consists of two secret key vectors $\mathbf{s}_1$ and $\mathbf{s}_2$, both Bruinderink and Pessl [GBP18] as well as Ravi et al. [RJH+18] have presented methods to sign arbitrary messages with knowledge of just $\mathbf{s}_1$. Thus, recovering $\mathbf{s}_1$ is sufficient to achieve an (equivalent) key recovery.

Algorithm 2 Dilithium Key generation

1: $\zeta \leftarrow \{0,1\}^{256}$ ▷ Sample random bytes
2: $(\rho, \rho', K) \in \{0,1\}^{256} \times \{0,1\}^{512} \times \{0,1\}^{256} \coloneqq \mathrm{H}(\zeta)$
3: $(\mathbf{s}_1, \mathbf{s}_2) \in S^{\ell}_{\eta} \times S^{k}_{\eta} \coloneqq \texttt{ExpandS}(\rho')$
4: $\mathbf{A} \in R^{k \times l}_q \coloneqq \texttt{ExpandA}(\rho)$
5: $\mathbf{t} \coloneqq \mathbf{A}\mathbf{s}_1 + \mathbf{s}_2$
6: $(\mathbf{t}_1, \mathbf{t}_0) \coloneqq \texttt{Power2Round}_q(\mathbf{t}, d)$
7: $tr \in \{0,1\}^{256} \coloneqq H(\rho \parallel \mathbf{t}_1)$
8: **return** $(pk \coloneqq (\rho, \mathbf{t}_1), sk \coloneqq (\rho, K, tr, \mathbf{s}_1, \mathbf{s}_2, \mathbf{t}_0))$

4 Prior Loop-Abort Faults on Fiat–Shamir with aborts Attacks

In [EFGT16], Espitau et al. demonstrate a loop-abort fault attack against Fiat–Shamir with aborts signature schemes. The attack is presented against BLISS, but a generalization to Dilithium is straightforward. In this section, we describe the initial attack against BLISS along with the shuffling countermeasure intended to mitigate loop-abort attacks.

The key idea of Espitau's attack is to induce a fault in the loop sampling the coefficients for the commitment vector $\mathbf{y}_1$. The coefficients are sampled in a sequential manner, starting with the coefficient of the lowest degree. A loop-abort yields a commitment polynomial $\mathbf{y}_1$ with an unexpectedly low degree. This enables an attacker to recover the secret key from a faulted signature $\mathbf{z}_1 = \mathbf{y}_1 + (-1)^b \mathbf{s}_1 c$ in a low-dimensional lattice via standard lattice reduction techniques [EFGT16].

The secret key can be recovered as follows. Given that the attacker induces a fault causing the termination of the sampling loop after n' out of n, ($n' << n$) iterations, the resulting commitment polynomial $\mathbf{y}_1$ is of degree at most $n' - 1$. The attacker also observes the generated signature and obtains the pair $(c, \mathbf{z}_1 = (-1)^b \mathbf{s}_1 c + \mathbf{y}_1)$. Assuming $b = 0$ without loss of generality, a crucial insight is that if we assume that c is invertible[1], then the vector $\mathbf{z}_1 c^{-1}$ is close to a sublattice of $\mathbb{Z}^n$. Consider the equation

$$\mathbf{z}_1 c^{-1} - \mathbf{s}_1 \equiv c^{-1}\mathbf{y}_1 \equiv \sum_{i=0}^{n'-1} y_{1,i} c^{-1} \mathbf{x}^i \mod q$$

[1] with high probability of $(1 - \frac{1}{q})^n$

Algorithm 3 Dilithium Signature generation

Input: A message μ, a secret key $(\mathbf{s}_1, \mathbf{s}_2)$
Output: A signature $\sigma{=}(\mathbf{z},\ \mathbf{h},\ c)$

$\mathbf{A} \in R_q^{k\times\ell} \coloneqq \texttt{ExpandA}(\rho)$
$\mu \in \{0,1\}^{512} \coloneqq \texttt{H}(tr \parallel M)$
$\kappa := 0, (\mathbf{z}, \mathbf{h}) := \perp$
$\rho' \in \{0,1\}^{512} \coloneqq H(K \parallel \mu)$ (or $\rho' \leftarrow \{0,1\}^{512}$ for randomized signing)
while $(\mathbf{z}, \mathbf{h}) = \perp$ **do**
  $\mathbf{y} \in \widetilde{S}^{\ell}_{\gamma_1} \coloneqq \texttt{ExpandMask}(\rho', \kappa)$
  $\mathbf{w} \coloneqq \mathbf{A}\mathbf{y}$
  $\mathbf{w}_1 \coloneqq \texttt{HighBits}_q(\mathbf{w},\ 2\gamma_2)$
  $\widetilde{c} \in \{0,1\}^{256} \coloneqq \texttt{H}(\mu||\mathbf{w}_1)$
  $c \in B_\tau \coloneqq \texttt{SampleInBall}(\widetilde{c})$
  $\mathbf{z} \coloneqq \mathbf{y} + c\mathbf{s}_1$
  $r_0 \coloneqq \texttt{LowBits}_q(\mathbf{w} - c\mathbf{s}_2\ ,\ 2\gamma_2)$
  if $\|\mathbf{z}\|_\infty \geq \gamma_1 - \beta$ or $\|\mathbf{r}_0\|_\infty \geq \gamma_2 - \beta$ **then**
    $(\mathbf{z}, \mathbf{h} := \perp)$
  else
    $\mathbf{h} \coloneqq \texttt{MakeHint}_q(-c\mathbf{t}_0\ ,\ \mathbf{w} - c\mathbf{s}_2 + c\mathbf{t}_0\ ,\ 2\gamma_2)$
    if $\|c\mathbf{t}_0\| \geq \gamma_2$ or the number of 1's in $\mathbf{h}$ is greater than ω **then**
      $(\mathbf{z}, \mathbf{h}) := \perp$
    end if
  end if
  $\kappa \coloneqq \kappa + l$
end while
return $\sigma := (\mathbf{z},\ \mathbf{h},\ c)$

and note that $\mathbf{s}_1$ is small, it thus follows that the vector $\mathbf{z}_1 c^{-1}$ is close to the sublattice $\mathcal{L}$ that is spanned by the vectors $w_i = c^{-1}\mathbf{x}^i \mod q$ for $i = 0, \ldots, n'-1$ and $q\mathbb{Z}^n$. The difference between $\mathbf{v} = \mathbf{z}_1 c^{-1}$ and the closest lattice point to $\mathbf{v}$ is exactly $\mathbf{s}_1$.

As the dimension of $\mathcal{L}$ is still n, this prevents us from directly applying lattice reduction algorithms. Instead, Espitau et al. propose to project the lattice as well as the target vector $\mathbf{z}_1 c^{-1}$ to a subset of the n dimensions. After projecting $\mathbf{z}_1 c^{-1}$ as well as the basis vectors of the lattice $\mathcal{L}$ to a subset of their rows, it still holds that the projected target vector is close to the projected lattice and the difference is the projected $\mathbf{s}_1$. If the projection is chosen such that the degree is low enough, we can recover a part of $\mathbf{s}_1$. By choosing multiple such projections we can eventually recover the entire secret polynomial and by that the entire secret key. Additionally, Espitau et al. propose two cryptographic countermeasures against their loop-abort faults attacks [EFGT16]:

The first proposed countermeasure is to shuffle the sampling order of $\mathbf{y}_1$'s coefficients. Note that the success of the attack by Espitau et al. is contingent on knowing the location of the zero-coefficients in $\mathbf{y}_1$. If the attacker does not know the indices of the zero coefficients, the lattice reduction technique is not applicable. Thus, by *shuffling* the sampled coefficients, Espitau et al. aim to prevent loop-abort fault attacks. The second countermeasure proposed by Espitau et al. is to only output signatures if the corresponding commitment is a high degree polynomial. This will thwart the lattice reduction attack, as this attack relies on a small lattice dimension. In the following sections, we show that the two countermeasures suggested by Espitau et al. are not sufficient to protect Fiat–Shamir with aborts implementations against loop-abort or zeroing faults.

5 Breaking the Shuffling Countermeasure via Integer Linear Programming

5.1 Main Idea

The primary focus of this paper is to demonstrate a loop-abort fault attack on implementations of Fiat–Shamir with aborts schemes that employ shuffling or low-degree checking countermeasures. In explaining the general methodology of our attack, we assume that the attacked scheme follows the general Fiat–Shamir with aborts structure. Recall that the commitment vector $\mathbf{y}$ and the secret key $\mathbf{s}$ are vectors of ℓ polynomials in R_q, i.e., $\mathbf{y} \in R_q^\ell$, $\mathbf{s} \in R_q^\ell$, and the signature is computed as $(\mathbf{z} = \mathbf{s}c + \mathbf{y}, c)$, where $c \in R$ is a challenge polynomial with small coefficients [Lyu09]. We assume that the coefficients of $\mathbf{y}$'s polynomial entries are sampled sequentially. We highlight the change to Section 4, which focused on BLISS, where the commitment vector $\mathbf{y}$ and the secret key $\mathbf{s}$ are vectors of integers.

Our approach is centered on introducing a loop-abort fault during the sampling of the commitment vector $\mathbf{y}$. As a result, most of the coefficients of the polynomials within the commitment vector are initialized to a fixed, yet recognizable value. After the shuffling process, the indices of the zeroed (or fixed) coefficients within the $\mathbf{y}$ vector are unknown to the attacker. However, the attacker can still leverage the fact that a significant portion of the $\mathbf{y}$'s coefficients are zeroed (or set to a known value). This knowledge can be utilized in an Integer Linear Program (ILP) to retrieve the secret key $\mathbf{s}$. To this end, we will draw inspiration from an ILP first presented in [MUTS22]. The ILP formulation has proven to be efficiently solvable in practice. Indeed, heuristically, the ILP's relaxation already provides a near-optimal solution in most cases. Section 6 gives detailed insights into the ILP's performance.

Attack Requirements. In summary, our attack has the following requirements. First, the attacker needs to be able to inject a loop-abort fault or a zeroing fault during Dilithium's signature generation. Additionally, the attacker must know the signature outputs, but does not need control over the signed messages. The attack only requires that at least one of the coefficients among $\mathbf{y}$'s polynomials is set to zero (or another known constant value). Importantly, the attacker does not need to know which specific coefficients are zeroed. The amount of faulted signatures required for key recovery depends on the number of zeroed coefficients per signature.

5.2 General Methodology

In this section, we describe the loop-abort fault key recovery attack against protected implementations of Fiat–Shamir with aborts signature schemes. For the purpose of this exposition, we therefore assume that the targeted implementation shuffles $\mathbf{y}$'s coefficients across all polynomials. When an attacker introduces a loop-abort fault during the sampling of $\mathbf{y}$, some coefficients among $\mathbf{y}$'s polynomials will be set to zero. The zeroed coefficients will have a uniformly random position due to the shuffling. As demonstrated later in this paper, our approach can be extended to encompass any scenario where the attacker knows that *some* (but not necessarily, *which*) of $\mathbf{y}$' coefficients are set to a fixed and known value.

We show that by partially faulting the generation of $\mathbf{y}$, the resulting faulty signature $\mathbf{z} = c\mathbf{s} + \mathbf{y}$ still reveals enough information to recover $\mathbf{s}$, even when the generation of $\mathbf{y}$ is shuffled. Intuitively, this is a consequence of the faulted $\mathbf{y}$ being sparse. The key idea is that a faulted signature $\mathbf{z} = c\mathbf{s} + \mathbf{y}$ with sparse vectors in $\mathbf{y}$ gives rise to a noisy equation system. Solving this equation system for $\mathbf{s}$ through an integer linear program reveals the secret key efficiently in practice, using only a few faulted signatures. We emphasize that

this is the case even if the attacker does not know the indices of the zeroed coefficients in $\mathbf{y}$.

In more detail, each equation of our equation system will express one coefficient of $\mathbf{z}$ as a linear dependency of the challenge c and the secret key vector $\mathbf{s}$. For a vector of polynomials $\mathbf{v} \in R^{\ell}$, let us denote by $v_{i,j}$ the j'th coefficient of the i'th polynomial. Then, we will express $z_{i,j}$ as

$$z_{i,j} = (c\mathbf{s})_{i,j}\,,$$

and if the corresponding coefficient $y_{i,j}$ is indeed 0 (e.g., because its sampling was skipped), then this equation holds true. We will refer to such equations as *correct* equations. If the corresponding coefficient is $y_{i,j}$ is not zero, then this equation is *incorrect*, as it is actually $z_{i,j} = (c\mathbf{s})_{i,j} + y_{i,j}$.

Assume now that an attacker managed to abort the sampling of $\mathbf{y}$ after the n'-iteration, with $n' < \frac{n}{2}$. Then, for each polynomial $\mathbf{y}_i, i = 1, \ldots, \ell$, it holds that the majority of its coefficients are set to zero. As a result, the relationship $z_{i,j} = (c\mathbf{s})_{i,j}$ holds for most of the coefficients, i.e., most of the formulated equations are *correct*. Therefore, identifying the correct secret key $\mathbf{s}$ from this set of noisy equations amounts to identifying the secret key $\mathbf{s}$ that maximizes the number of fulfilled equations. In SAC'2022, Marzougui et al. [MUTS22] presented a power side-channel attack against Dilithium, exploiting a leakage on the commitment vector $\mathbf{y}$'s coefficients to recover the secret key $\mathbf{s}$. To do so, they also collect the leakages into a noisy equation system; to recover the $\mathbf{s}$ from this noisy equation system, they formulate an ILP to find an $\mathbf{s}$ that maximizes the number of fulfilled equations. We adapt their method to recover $\mathbf{s}$ from the faulty signatures.

Before we describe the ILP that we will use to solve for the secret key, we present an additional observation that allows us to reduce the ILP's dimension. To this end, observe that we can split the given problem into ℓ separate sets of equations, one for each polynomial in the vector $\mathbf{s}$. Note that

$$c\mathbf{s} = \begin{pmatrix} c \cdot (\mathbf{s})_1 \\ c \cdot (\mathbf{s})_2 \\ \vdots \\ c \cdot (\mathbf{s})_{\ell} \end{pmatrix}.$$

To obtain $c\mathbf{s}$, we multiply c with each polynomial in the vector $\mathbf{s}$ independently. Note that $\mathbf{s} \in R^{\ell}$ consists of ℓ polynomials, each with n coefficients, while $c \in R$ is exactly one polynomial. One coefficient $z_{i,j} = y_{i,j} + (c\mathbf{s})_{i,j}$ is consequently influenced only by the i^{th} secret key polynomial $\mathbf{s}_i$. As a result, after faulting multiple signatures, we can create ℓ independent equation systems (one for each polynomial in $\mathbf{s}$) and solve for each polynomial of $\mathbf{s}$ separately.

We proceed as follows. For each equation system, let $L = ((z_{i,j}^{(1)} = c^{(1)}s^*), \ldots, (z_{i,j}^{(|L|)} = c^{(|L|)}s^*))$ denote the list of collected equations, where by $s^* \in R_q$ we denote the secret key polynomial we are currently solving for and by $\mathbf{z}^m, \mathbf{y}^m, c^m$ we denote the signature output, commitment vector, and challenge polynomial of the signature corresponding to the m'th collected equation respectively. We derive from the list of collected equations a vector $\mathbf{z}^* \in \mathbb{Z}^{|L|}$, where the m'th entry of $\mathbf{z}^*$ contains the signatures' coefficients $z_{i,j}^m$ and a matrix $\mathbf{C}^{|L| \times n}$, derived from the challenge polynomials c. Specifically, the m'th row of $\mathbf{C}$, denoted by $\mathbf{C}_m$, is constructed such that the dot product $\mathbf{C}_m \mathtt{vec}(s^*)$ equals $\mathbf{z}_m$.

We are now in a position to formulate the following Integer Linear Program to recover the secret key polynomial s^*, which we state in Figure 2.

$$
\begin{array}{lll}
\text{maximize} & \sum_{m=1}^{|L|} x_m & \\
\text{subject to} & & \\
& \mathbf{z}^*_m - \mathbf{C}_m \mathtt{vec}(s^*) \leq K \cdot (1 - x_m) & \forall m \in \{1, \ldots, |L|\} \quad (1) \\
& \mathbf{z}^*_m - \mathbf{C}_m \mathtt{vec}(s^*) \geq -K \cdot (1 - x_m) & \forall m \in \{1, \ldots, |L|\} \quad (2) \\
& x_m \in \{0, 1\} & \forall m \in \{1, \ldots, |L|\} \quad (3) \\
& s^*_j \in \{-\eta, \ldots, \eta\} & \forall j \in \{1, \ldots, n\} \quad (4)
\end{array}
$$

Figure 2: The ILP formulation used for recovering a Fiat–Shamir with aborts secret key polynomial from a noisy equation system.

Constraints (1) and (2) ensure that for $x_m = 1$ we have $\mathbf{z}^*_m - \mathbf{C}_m \mathtt{vec}(s^*) = 0$. This is canonically known as the big-M method — we choose K as the maximum possible distance between $\mathbf{z}^*_m$ and $\mathbf{C}_m \mathtt{vec}(s^*)$. Constraint (3) ensures that x_m will be a binary decision variable and constraint (4) ensures that each coefficient of the guessed secret key s^* is an integer in the appropriate range.

If the loop-abort fault has been introduced sufficiently early and with enough faulted signatures, this ILP can be solved efficiently in practice. The next section will detail the application of this attack to Dilithium.

Alternative key recovery method — LWE without modular reductions. The problem of recovering a secret key polynomial s^* from a system of linear equations $\mathbf{z}^* = \mathbf{C}s^* + \mathbf{e}$ can also be viewed as an LWE without modular reduction problem. To see why, note that it holds that $\|cs^* + \mathbf{y}\|_\infty < q$, and thus there are no modular reductions involved in the collected equations. Bootle et al. [BDE+18] show that the "LWE without modular reduction" problem can be solved with least-squares and rounding. They prove that the solution candidate obtained by least-squares and rounding converges to the correct solution if the error distribution follows a Gaussian distribution. However, we found that the least-squares method could not match the ILP-based method's performance in our experiments. For various instances, the least-squares method yielded an incorrect solution, while the ILP-based method was able to recover the correct secret key.

5.3 Attacking the Protected Implementation of Dilithium

To adapt the presented attack to Dilithium, we need to a) specify the appropriate location for the fault injection and b) adapt our ILP formulation to recover the secret key from the faulty signatures. Recall that Dilithium, following a lattice-based Fiat–Shamir with aborts structure, uses a secret key $\mathbf{s}_1$ and a commitment vector $\mathbf{y}$, both vectors of polynomials in the ring $R_q = \mathbb{Z}_q[X]/(X^n + 1)$. The coefficients of the secret key polynomials are all in the small range of $\{-\eta, \ldots, +\eta\}$, while $\mathbf{y}$'s polynomial coefficients are distributed uniformly in the range of $\{-\gamma_1 + 1, \ldots, \gamma_1\}$. The signature is then calculated as $z = c\mathbf{s}_1 + \mathbf{y}$, where the challenge polynomial c is derived from the message. By employing the ILP, we will be able to recover the secret key $\mathbf{s}_1$. In doing so, we have achieved an (equivalent) key recovery, as the knowledge of $\mathbf{s}_1$ is sufficient to sign arbitrary messages.

Fault Injection Site. To launch the ILP-based key recovery attack against Dilithium, the attacker needs to interfere with $\mathbf{y}$'s polynomial coefficients, so that an abnormal number of coefficients in $\mathbf{y}$'s polynomials are set to zero. This can be, again, realized through a loop-abort fault during the sampling of $\mathbf{y}$'s coefficients. In more detail, during the Dilithium signing process, the vector $\mathbf{y}$ is sampled from an extensible output function (XOF). The XOF is seeded with an initial randomness seed ρ' and nonce κ. In Dilithium's

deterministic version ρ' is sampled as from a hash derived from the message and some randomness generated during key generation, while in the non-deterministic version ρ' is sampled from system randomness. In the non-deterministic version, calculating a signature for the same message twice will therefore lead to different signatures. The XOF's output is unpacked into ℓ polynomials, coefficient-by-coefficient. For each coefficient, the `ExpandMask` function converts a fixed number of bytes from the buffer into an integer in the range $\{-\gamma_1 + 1, \ldots, \gamma_1\}$. Algorithm 4 lists the pseudocode of this sampling. Algorithm 5 lists a shuffled version, thwarting the attack by Espitau et al. by shuffling the coefficients of $\mathbf{y}$ uniformly at random across all of $\mathbf{y}$'s coefficients. The non-deterministic version is specifically designed to complicate fault injection and side-channel attacks. Nevertheless, our attack can be carried out on both versions of Dilithium, i.e., deterministic and non-deterministic. The sampling of $\mathbf{y}$ serves as a perfect location for the desired loop-abort

Algorithm 4 The `ExpandMask` Function

1: $buf \coloneqq \mathrm{XOF}(\rho', \kappa)$
2: **for** $i \coloneqq 0$ to ℓ **do**
3: **for** $j \coloneqq 0$ to n **do**
4: $y_{i,j} \coloneqq \texttt{ConvertBuf}(buf)$ ▷ Convert some bytes of buf to an integer in range $\{-\gamma_1 + 1, \ldots, \gamma_1\}$
5: **end for**
6: **end for**
7: **return** $\mathbf{y}$

Algorithm 5 The Shuffled `ExpandMask` Function

1: $\mathbf{y} \coloneqq \texttt{ExpandMask}(\rho', \kappa)$
2: $\mathbf{y} \coloneqq \texttt{Shuffle}(\mathbf{y})$ ▷ Shuffle *all* coefficients of $\mathbf{y}$, also across polynomials using Fisher-Yates shuffle, seeded with randomness ρ' [Knu97].
3: **return** $\mathbf{y}$

fault attack. If an attacker causes an early loop-abort of the outer or inner sampling loop in Line 2 and 3, respectively, then the rest of $\mathbf{y}$'s polynomial coefficients will be set to a fixed value. Without the shuffling mechanism, such a fault enables an attacker to launch the lattice-reduction attack. In case the shuffled version of `ExpandMask` is used, the coefficients will be shuffled across $\mathbf{y}$ uniformly at random. However, the ILP-based attack presented in this paper can still recover $\mathbf{s}_1$.

ILP Key Recovery For Dilithium. An adaption of the ILP described in Section 5 (Figure 2) to Dilithium is straightforward, and we can again draw from the ILP formulation presented by Marzougui et al. [MUTS22]. A collection of faulted signatures of the form $\mathbf{z} = c\mathbf{s}_1 + \mathbf{y}$, where $\mathbf{y}$ consists of sparse polynomials, gives rise to a noisy system of linear equations from which we can recover $\mathbf{s}_1$. As described in Figure 2, we derive ℓ equation systems from the faulted signatures, each containing equations of the form $z_{i,j} = (c\mathbf{s}_1)_{i,j} + y_{i,j}$. For each equation system, we derive a matrix $\mathbf{C} \in \mathbb{Z}^{|L| \times n}$ and a vector $\mathbf{z}^* \in \mathbb{Z}^{|L|}$ (where L is the list of equations) and solve for the secret key polynomial $s^* = (s_1)_i, i = 1, \ldots, \ell$.

Before we describe the ILP formulation, we present an additional optimization to increase the attack's performance and reduce the number of required signatures. To do so, let us state two observations about the attack's correctness and the performance of the ILP solver. First, to guarantee the correctness of the ILP output, i.e., that the ILP indeed outputs the correct secret key $\mathbf{s}_1$, the majority of the equations needs to be correct. Second, through empirical observations, we found that the ratio of correct vs. incorrect equations

also influences an ILP solver's performance, where more correct equations result in a faster solving time. We thus employ the following optimization when attacking Dilithium.

To improve the ratio of correct vs. incorrect equations, we make use of the fact that both c's as well as $\mathbf{s}_1$'s polynomial coefficients are small; thus, $c\mathbf{s}_1$'s coefficients must be small as well. In more detail, it holds that $|(c\mathbf{s}_1)_{i,j}| \leq \beta$, where $\beta = \tau \cdot \eta$. Thus, assuming that $y_{i,j} = 0$, it holds that $|\mathbf{z}_{i,j}| = |\mathbf{y}_{i,j} + (c\mathbf{s}_1)_{i,j}| \leq \beta$ as well. We can therefore dismiss the possibility that a certain coefficient $\mathbf{y}_{i,j} = 0$ if the corresponding coefficient $|\mathbf{z}_{i,j}|$ exceeds β.

We further observe that each coefficient $(c\mathbf{s}_1)_{i,j}$ can be approximated by a normal distribution. Recall that c is a sparse vector with τ coefficients being -1 or 1 and the rest zero. Therefore, the coefficients of the polynomial $c\mathbf{s}_1$ can be modeled as the sum of τ i.i.d. random variables, each variable being uniformly distributed over the range $[-\eta, \ldots, \eta] \cap \mathbb{Z}$. By a central limit theorem argument, this sum is close to a normal distribution with mean 0 and variance $\sigma^2 = \frac{(2\cdot\eta)^2-1}{12\cdot\tau}$. If we want to further reduce the number of incorrect equations at the expense of potentially only a few more signatures, we can therefore dismiss coefficients for which it holds that $|\mathbf{z}_{i,j}| > 2 \cdot \sigma$, since most of $|(c\mathbf{s}_1)_{i,j}|$ coefficients will fall in that range with high probability.

To reduce the number of incorrect equations, we therefore only insert an equation $z_{i,j} = (cs)_{i,j}$ into our set of equations if $|z_{i,j}|$ is below or equal a pre-defined threshold. We recommend setting this threshold to either β (to avoid any missed correct equations) or $2 \cdot \sigma$ (to minimize the number of incorrect equations at the expense of more signatures). Otherwise, if $|z_{i,j}|$ is larger than this threshold, we assume that $\mathbf{y}_{i,j} \neq 0$ and do not add an equation to our equation system.

We are now in a position to describe an end-to-end loop-abort fault attack against Dilithium. Figure 3 describes the adapted ILP.

$$
\begin{aligned}
&\text{maximize} && \sum_{m=1}^{|L|} x_m && && \\
&\text{subject to} && && && \\
& && \mathbf{z}^*_m - \mathbf{C}_m \mathtt{vec}(\mathbf{s}^*) \leq 2\beta + \gamma_1 \cdot (1 - x_m) && \forall m \in \{1, \ldots, |L|\} && (1)\\
& && \mathbf{z}^*_m - \mathbf{C}_m \mathtt{vec}(\mathbf{s}^*) \geq -(2\beta + \gamma_1) \cdot (1 - x_m) && \forall m \in \{1, \ldots, |L|\} && (2)\\
& && x_m \in \{0, 1\} && \forall m \in \{1, \ldots, |L|\} && (3)\\
& && s^*_j \in \{-\eta, \ldots, \eta\} && \forall j \in \{1, \ldots, n\} && (4)
\end{aligned}
$$

Figure 3: The ILP formulation used for recovering a Dilithium secret key polynomial from a noisy equation system.

First, the attacker faults multiple signature generations and derives ℓ noisy equation systems from the resulting outputs. In doing, so, the attacker filters out potentially incorrect equations by only dismissing coefficients for which it holds that $|z_{i,j}| > 2 \cdot \sigma$. Next, the attacker solves ℓ ILPs. The attacker can then verify if a key guess $\mathbf{s}_1$ is correct. If the key is incorrect or the ILP solvers do not terminate within the allotted time, the attacker can collect more faulted signatures and repeat the attack. We evaluate the performance of the ILP and the performance in relation to the number of zeroed coefficients in Section 6.

6 Simulation and Evaluation of the Attack

We evaluate the efficacy of our proposed key recovery method through extensive simulations. The purpose of this theoretical evaluation is to determine the minimum number of faulty

Algorithm 6 Key Recovery via Integer Linear Programming

Input: A list $\mathcal{L}$ of potentially faulty signatures
$\mathcal{L} = ((\mathbf{z}^1 = (c^1\mathbf{s}_1) + \mathbf{y}, \mathbf{h}^1, c^1), \ldots, (\mathbf{z}^{|\mathcal{L}|} = (c^{|\mathcal{L}|}\mathbf{s}_1) + \mathbf{y}, \mathbf{h}^{|\mathcal{L}|}, c^{|\mathcal{L}|}))$
Output: The secret key $\mathbf{s}_1$

$L \coloneqq$ A list of ℓ lists, where list L_i contains equations for secret key polynomial s_i
for $(m, i, j) \in ((1 \ldots |\mathcal{L}|) \times (1 \ldots \ell) \times (1 \ldots n))$ **do**
  if $|z^m_{i,j}| \leq \beta$ (or $\leq 2\sigma$) **then**
    Add equation $z^m_{i,j} = c^m\mathbf{s}_1$ to list L_i
  end if
end for
for $i = 1, \ldots, \ell$ **do**
  Derive from the equations in L_i the vector $\mathbf{z}^* \in \mathbb{Z}^{|L_i|}$ and the matrix $\mathbf{C} \in \mathbb{Z}^{|L_i| \times n}$.
  Solve ILP (as described in Figure 3) to recover the secret key polynomial $\mathbf{s}_i$ from the equation system $\mathbf{z}^* = \mathbf{C}\mathtt{vec}(\mathbf{s}_i)$.
end for
return The secret key $\mathbf{s}_1$

signatures required for successful key recovery. We also evaluate how the number of zeroed coefficients per signature and the targeted Dilithium security level influence the attack's complexity and success.

Evaluation Setup. Algorithm 7 summarizes our evaluation method. The evaluation assumes that the attacker can fault multiple signatures. For each signature, the attacker injects a loop-abort fault during sampling of $\mathbf{y}$, zeroing some of $\mathbf{y}$'s coefficients. We assume that the zeroed coefficients are scattered over all polynomials in $\mathbf{y}$. The attacker then receives the faulted signature, and knows that some of $\mathbf{y}$'s coefficients have been set to zero, but does not know which.

The evaluation varies three factors that influence the success rate of our attack. First, the number of coefficients in the commitment vector $\mathbf{y}$ that an attacker can set to zero within each faulted signature. We denote this number by t. The number of zeroed coefficients is influenced by the attacker model. An early loop-abort in sampling $\mathbf{y}$ will result in more zeroed coefficients, but it is potentially harder to implement. Faulting almost all of $\mathbf{y}$'s coefficients could also be prevented by a countermeasure that checks if $\mathbf{y}$ is of abnormally low degree before outputting the signature. Second, we vary the number of faulted signatures an attacker collects before attempting key recovery. The assumed attacker continues collecting faulted signatures until they have enough information to derive at least M correct equations per secret key polynomial. Equivalently, the assumed attacker attempts key recovery once they have zeroed at least a total of M coefficients per polynomial in $\mathbf{y}$, aggregated over all signatures. The third factor is the security level of the attacked Dilithium implementation.

Note that the number of signatures used depends on all three parameters. To evaluate the minimum number of faulted signatures required for the successful secret key recovery, we increment the value of M while keeping t and Dilithium's security level constant. We then increase M until the ILP can recover the secret key in less than five minutes.

Results. Figure 4 depicts our evaluation results. We executed the simulation on an Intel(R) Xeon(R) CPU E7-4870 running Ubuntu 20.04, and used the Gurobi solver suite to solve the ILP instances [Gur23].

For all security levels, the presented method is able to recover the secret key $\mathbf{s}_1$ with enough faulted signatures. Naturally, the number of required faulted signatures decreases when the number of faulted coefficients per signature increases. The simulations also

Algorithm 7 Simulation of the key recovery via ILP

Input: An assumed number of zeroed coefficients t per faulted signature and a NIST security level for Dilithium, defining the parameters k, ℓ, τ, η.
Output: The number of used signatures for successful key recovery, i.e., $|\mathcal{L}|$.

$M \coloneqq n - 1$
repeat
  $\mathcal{L} \coloneqq$ Empty list which will contain the faulted signatures
  $M \coloneqq M + 1$
  repeat
    Generate a faulted signature for a random message and add the signature to $\mathcal{L}$. We calculate the faulted signature by setting t *random* coefficients in $\mathbf{y}$ to 0 before computing $\mathbf{z} = c\mathbf{s}_1 + \mathbf{y}$.
  until We have aggregated at least M zeroed coefficients for each polynomial in $\mathbf{y}$
  Attempt to recovery the secret key $\mathbf{s}_1$ using Algorithm 6 with a time limit of 5 minutes for all ILP solvers.
until Recovery Successful
return The number of used signatures for successful key recovery, i.e., $|\mathcal{L}|$.

revealed the following insight. Asymptotically, ILPs have exponential complexity, but the ILP's rational relaxation often already provides a near-optimal solution. This enables us to recover the key in practice. An increase in the total number of zeroed coefficients (and, in turn, correct equations) correlates with a lower run time in practice, as can also be derived from Figure 4.

The evaluation shows that the ILP method can break two cryptographic countermeasures proposed by Espitau et al. First, shuffling the sampling order of $\mathbf{y}$'s coefficients does not mitigate the ILP attack. This is because the ILP attack does not rely on knowing which exact coefficients have been set to zero. Rather, it is sufficient to provide it with equation systems that are derived from signatures where some of the coefficients are set to zero. Moreover, a countermeasure that ensures that the polynomials in $\mathbf{y}$ are not of low degree also falls short since fixing just *one* coefficient to zero is sufficient to recover the secret key, given enough signatures. As a result, more sophisticated, potentially expensive, countermeasures are needed.

7 End-To-End Attack Proof-of-Concept

In this section, we present an end-to-end proof-of-concept attack on a protected implementation of Dilithium. We launch a fault-injection attack on a modified Dilithium reference implementation, as submitted to the NIST call for proposal [BDK+]. We modified the implementation to additionally implement the shuffling countermeasure. We then executed the signing process on an ARM Cortex M4, mounted on a ChipWhisperer target board [OC14]. We describe our experimental setup, including our workbench, and outline the details of our attack. We will make the implementation of the attack, along with the generated faulty signatures and the parameters used for the glitch attack, publicly available.

7.1 Experimental Workbench

Our fault injection setup comprises an STM32F4 target board with 1 MB of Flash memory and 192 KB of RAM. The target board is mounted on the ChipWhisperer UFO board with three 20-pin female headers into which the target board fits. We modified the Dilithium reference implementation to shuffle $\mathbf{y}$'s coefficients and execute the modified signature

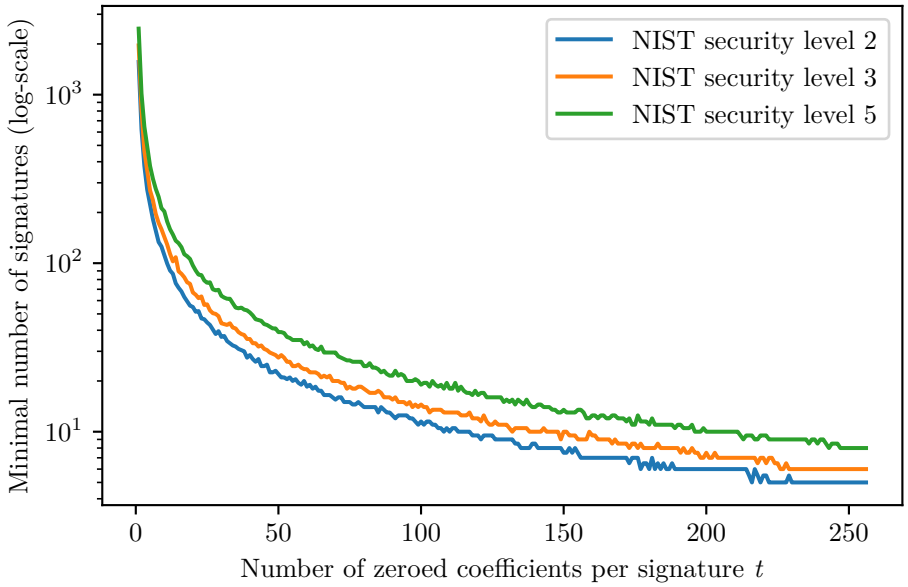

Figure 4: Minimal amount of faulted signatures required to recover $\mathbf{s}_1$ through Algorithm 6 in under 5 minutes, depending on the number zeroed coefficients per signature t and the NIST security level. The amount of required faulted signatures is calculated as described in Algorithm 7.

generation on the target board. We configured Dilithium to run at security level 2, as described in the Dilithium specification [BDK+21]. We collect multiple faulted signatures and recover the secret key using Algorithm 6. We use the same private key for all signatures, which we generated offline prior to the experiment and then flashed onto the target board.

Dilithium Modification. The Dilithium reference implementation samples $\mathbf{y}$ as described in Algorithm 4. It first samples a random byte-buffer through an XOF, and then converts this byte-buffer to $\mathbf{y}'s$ coefficients in two nested loops. The outer loop of the sampling process, displayed in Listing 1, sequentially calls the unpacking function for each polynomial $\mathbf{y}_i, i \in 1, \ldots, \ell$. Listing 2 represents the inner loop. It focuses on unpacking a byte buffer into $n = 256$ coefficients for a particular polynomial. To shuffle the coefficients, we introduce the `polyvecl_shuffle` function, which shuffles all of $\mathbf{y}$'s coefficients after they have been unpacked. We give our implementation in Listing 3 in the Appendix. The signature generation code then first samples the coefficients by calling `polyvecl_uniform_gamma1` and shuffles them afterwards by calling `polyvecl_shuffle`.

7.2 Fault Injection Attack

To skip the sampling loop we execute a clock-glitching attack, leveraging the ChipWhisperer. The ChipWhisperer is used to generate the base clock of the microcontroller, and the clock frequency was set to 7,327 MHz. When a GPIO pin is activated, the ChipWhisperer system modifies the clock pulse, causing the executing microcontroller to skip instructions. The GPIO pin is triggered by the code under test itself. To facilitate this triggering mechanism, we have included additional instructions within the code. At the beginning of the `polyz_unpack` function we added instructions to raise the GPIO pin and added

Listing 1 The functions that samplee ℓ polynomials sequentially from a seed.

```
#define POLY_UNIFORM_GAMMA1_NBLOCKS ((POLYZ_PACKEDBYTES +  \
                    STREAM256_BLOCKBYTES - 1)/STREAM256_BLOCKBYTES)
void polyvecl_uniform_gamma1(polyvecl *v, const uint8_t seed[CRHBYTES],
uint16_t nonce) {
  unsigned int i;
  for(i = 0; i < L; ++i)
    poly_uniform_gamma1(&v->vec[i], seed, L*nonce + i);
}

void poly_uniform_gamma1(poly *a,
                         const uint8_t seed[CRHBYTES],
                         uint16_t nonce)
{
  uint8_t buf[POLY_UNIFORM_GAMMA1_NBLOCKS*STREAM256_BLOCKBYTES];
  stream256_state state;
  stream256_init(&state, seed, nonce);
  stream256_squeezeblocks(buf, POLY_UNIFORM_GAMMA1_NBLOCKS, &state);
  polyz_unpack(a, buf);
}
```

Listing 2 The `polyz_unpack` Dilithium level 2 reference implementation including the triggering code.

```
void polyz_unpack(poly *r, const uint8_t *a) {
 unsigned int i;
 trigger_high();
   for(i = 0; i < N/4; ++i) {
     r->coeffs[4*i+0]  = a[9*i+0];
     r->coeffs[4*i+0] |= (uint32_t)a[9*i+1] << 8;
     r->coeffs[4*i+0] |= (uint32_t)a[9*i+2] << 16;
     r->coeffs[4*i+0] &= 0x3FFFF;

     r->coeffs[4*i+1]  = a[9*i+2] >> 2;
     r->coeffs[4*i+1] |= (uint32_t)a[9*i+3] << 6;
     r->coeffs[4*i+1] |= (uint32_t)a[9*i+4] << 14;
     r->coeffs[4*i+1] &= 0x3FFFF;

     r->coeffs[4*i+2]  = a[9*i+4] >> 4;
     r->coeffs[4*i+2] |= (uint32_t)a[9*i+5] << 4;
     r->coeffs[4*i+2] |= (uint32_t)a[9*i+6] << 12;
     r->coeffs[4*i+2] &= 0x3FFFF;

     r->coeffs[4*i+3]  = a[9*i+6] >> 6;
     r->coeffs[4*i+3] |= (uint32_t)a[9*i+7] << 2;
     r->coeffs[4*i+3] |= (uint32_t)a[9*i+8] << 10;
     r->coeffs[4*i+3] &= 0x3FFFF;

     r->coeffs[4*i+0] = GAMMA1 - r->coeffs[4*i+0];
     r->coeffs[4*i+1] = GAMMA1 - r->coeffs[4*i+1];
     r->coeffs[4*i+2] = GAMMA1 - r->coeffs[4*i+2];
     r->coeffs[4*i+3] = GAMMA1 - r->coeffs[4*i+3];
   }
  trigger_low();
}
```

instructions to lower it again at the end of the `polyz_unpack` function (see Listing 2). As a result, the GPIO pin remains high throughout the execution of the coefficient sampling/unpacking loop.

The clock modification performed by the ChipWhisperer system occurs within a single clock cycle. The ChipWhisperer modifies this clock as follows. During the targeted clock

period, the ChipWhisperer lowers the rising clock edge for `width` percent of a clock period, after leaving the rising edge unmodified for `offset` percent of a clock period. The attacker needs to adjust the parameters `width` and `offset` according to their specific objectives. Figure 5 illustrates the clock modification.

In order to determine the most effective glitching parameters for skipping the sampling instructions, we swept the parameter search space. This search revealed a `width` of 1.562500% and an `offset` of 0.390625% to be optimal for our attack. In order to target specific iterations of the unpacking loop in the `polyz_unpack` function, we additionally utilized the ChipWhisperer's `ext_offset` parameter. The `ext_offset` parameter defines how many clock cycles the ChipWhisperer should wait before injecting the clock skew (glitch). We were able to induce a loop-abort after the i-th iteration of the sampling loop by using an `ext_offset` of $55 + 62 \cdot (i - 1)$ with $i = 1, 2, \ldots, 63$.

We only glitched the first execution of the `polyz_unpack` function. As a result, we zeroed coefficients only in the first polynomial of $\mathbf{y}$. Due to the shuffling mechanism however, the zeroed coefficients of the first polynomial $\mathbf{y}_1$ are spread across all $\mathbf{y}_i$, with $i = 1, \ldots, \ell$. This enables us to recover all secret key polynomials of $\mathbf{s}_1$.

To test the efficacy of our attack, we conducted the following experiment.

- We repeatedly generate signatures on the ChipWhisperer for randomly generated messages. During signature generation, we inject a clock-glitching fault if the GPIO pin of the target board is high. We record the (potentially) faulty signature and whether the fault was successful or not.

- After each signature, we collect the faulted signatures outputs so far and try to recover the secret key via Algorithm 6. If the algorithm cannot recover the key in 5 minutes, we generate more faulty signatures and try again.

To simulate different (weaker) attacker capabilities, we conduct multiple experiments, varying the number of coefficients whose sampling the attacker can skip. We do so by delaying the point in time at which we introduce a fault. For example, if we want to skip the sampling of 4 coefficients per signature, we only inject a fault in the last iteration of the sampling loop in the `polyz_unpack` function. To target a specific iteration of the *polyz_unpack* loop, we time the fault appropriately by adjusting the *ext_offset* parameter. Doing so makes the attacker artificially *weaker*, as a real-world attacker would introduce the fault as early as possible.

7.3 Experimental Results

We summarize our results in Table 1. For each attack, we record the total execution time, accounting for both the signature collection as well as the time required to run Algorithm 6.

We were able to skip 252 coefficients per faulted signature, recovering the secret key in under three minutes, using only 5 faulted signatures, and generating 53 signatures in total.

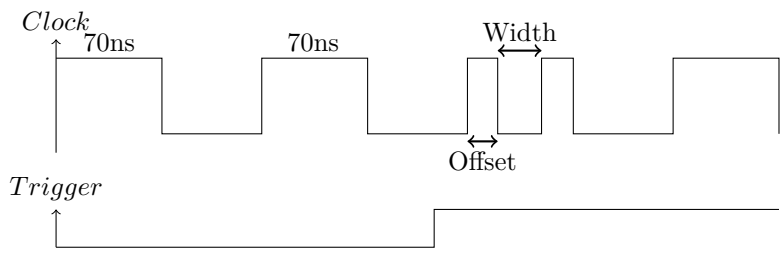

Figure 5: An illustration of the ChipWhisperer's clock modification.

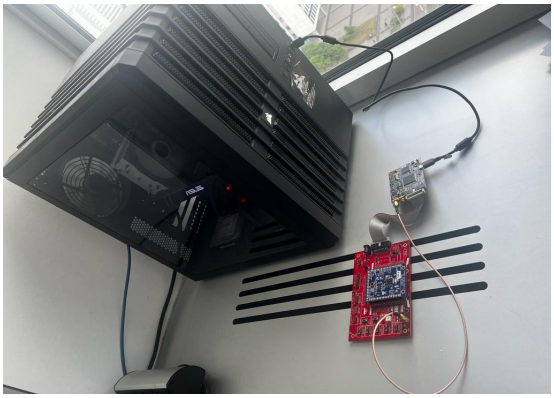

Figure 6: Experimental workbench used for our fault injection attack; it contains an STM32F4 target board mounted on the UFO board (red), a ChipWhisperer CW308, two SMA cables, and a USB cable.

If we artificially weaken the attacker (skipping fewer coefficients per signature), the attacker can still recover $\mathbf{s}_1$. The weakest attacker, skipping only one iteration of the `polyz_unpack` loop, can recover $\mathbf{s}_1$ after generating 3466 signatures, out of which 225 were faulty. The whole attack can be launched, end-to-end, in under four hours. Notably, for all equations systems, the key recovery algorithm takes less than one minute, heuristically indicating that the ILP relaxation already provides the correct secret key.

Table 1: Number of failures and successes when executing the loop abort attack on the shuffled implementation of Dilithium. The amount of skipped coefficients is always a multiple of 4 as the Dilithium reference implementation samples 4 coefficients per iteration. A failure can either be due to the glitch being unreliable or due to a successfully faulted signature being rejected. The time measurements are given in the hour:minute format.

# Skipped coefficients	# Faulted Signatures	# Fault Failures	# Executed Signatures	Total Execution Time (hh:mm)	Key Recovery Time (hh:mm)	Signature Generation Time (hh:mm)
4	225	3466	3691	03:14	< 00:01	03:14
8	132	2190	2322	02:02	< 00:01	02:02
12	102	1923	2025	01:46	< 00:01	01:46
16	75	1614	1689	01:29	< 00:01	01:29
20	61	982	1043	00:56	< 00:01	00:56
252	5	48	53	00:03	< 00:01	00:02

8 Discussion

The presented attack is contingent only on the presence of zeroed or fixed value coefficients in the commitment vector $\mathbf{y}$. This can be achieved through, for example, introducing a loop-abort fault during the sampling of $\mathbf{y}$'s coefficients. Alternatively, our attack can also be realized through a *zeroing attack*, directly targeting memory cells and fixing their value to zero. Zeroing attacks are typically targeted at hardware implementations. To execute a zeroing attack on Dilthium, an attacker can, for example, zero the random values generated by the XOF and stored in BRAM at a specific point in time.

8.1 Applicability to Other Implementations

Our attack generalizes to other implementations and can bypass multiple side-channel and fault-injection countermeasures, as the required information leak is small. For instance,

our attack can also subvert the non-deterministic version of Dilithium.

Previous works have focused on optimizing the performance and security of Dilithium against side-channel attacks [KPR+, ZZW+21, LSG22, ZHL+21, GJCJ22, MGTF19]. However, none have examined the vulnerability of the scheme to fault injection attacks. Consequently, all of these implementations remain vulnerable to the presented fault injection attack.

Masked implementations might also still be vulnerable to our loop-abort fault injection attack, as masking in and of itself is not a sufficient countermeasure against the proposed loop-abort attack. In masked implementations sensitive information is stored and processed in different shares, adding resilience against side-channel analysis. For instance, Migliore et al. [MGTF19] propose a masked implementation of Dilithium that increases resiliency against (power) side-channel attacks. This implementation, however, still samples the coefficients of $\mathbf{y}$ in a sequential manner. As a result, a loop-abort fault would still result in $\mathbf{y}$'s coefficients being zero or set to a fixed value, enabling the attacker to recover the secret key.

8.2 Countermeasures

Our attack appears hard to mitigate using purely cryptographic defenses. Alternatively, protected implementations can resort to redundant computation or checksumming to detect fault injection attempts (and abort execution accordingly, withholding the signature output).

A promising method to detect fault injection attempts relies on Concurrent Error Detection (CED) schemes. In CED schemes, we execute both the target algorithm a and a designated predictor a'. The result of the real algorithm (a) and the predictor (a') are checked in another module (c) to detect possible errors before transmitting the output [YW06, AMR+20]. This countermeasure poses an effective defense against fault-injection attacks [AMR+20]. However, designing a well-performing predictor (a') for Dilithium can be challenging and expensive.

To implement CED for Dilithium, one could execute the signature generation for the same message twice and compare the results. However, this method is not applicable to the non-deterministic version of Dilithium. Additionally, it adds a considerable overhead, making this countermeasure detrimental to performance. Instead, a protected implementation could focus on only re-executing the unpacking process and comparing the unpacked coefficients.

One could also implement a loop iteration counter within the nested unpacking loops and compare the expected value with the actual value after the loops have been executed. A similar way to detect skipped sampling instructions is by counting the cycles during program execution to effectively identify any injected faults. We can track the number of cycles taken for the unpacking routine to track whether a fault is injected during the unpacking of the vector $\boldsymbol{y}$. For valid signatures that are not targeted by our fault injection attack, the cycle count should remain consistent with the expected number of cycles. However, both these countermeasures do not protect against zeroing faults that directly target the memory.

To specifically defend the implementation against zeroing attacks, one can store the binary inverse and the original value of the vector $\mathbf{y}$ redundantly and compare both values after unpacking to verify data integrity. Another approach is to calculate the checksum of the randomness buffer acquired through the XOF and store this checksum together with the buffer. When the randomness buffer is read, its checksum should also be verified to detect any potential attack where the random values stored in memory are set to zero.

9 Conclusion

Our paper presents a powerful key recovery fault injection attack against Fiat–Shamir lattice-based signatures, which defeats several countermeasures proposed in previous work and appears hard to thwart using purely cryptographic defenses. We validated our attack experimentally in a practical end-to-end demonstration against a protected ARM Cortex M4 implementation of NIST selected scheme Dilithium, recovering the full key from just a handful of faulty signatures.

Our analysis of the attack highlights a number of strengths of our approach:

- First, our attack bypasses the two cryptographic fault-injection countermeasures proposed by Espitau et al. [EFGT16], namely shuffling the order in which the coefficients of the commitment vector $\mathbf{y}$ are sampled, as well as rejecting low-degree polynomials. Moreover, since our attack successfully breaks the scheme given faulty signatures with as little as a single zeroed coefficient, it seems difficult to protect against using any sort of variant of these approaches.
- Second, our attack breaks not only the deterministic version of Dilithium but also the non-deterministic variant, which is specifically designed to provide greater protection against side-channel [SBB+18] and fault-injection attacks [PSS+18]. Unlike our attack, differential fault attacks cannot be launched against the non-deterministic variant, as they require two signatures on the same message with the same challenge.
- Third, our attack easily extends to cases where the coefficients of an uninitialized polynomial $\mathbf{y}$ are not zero, but rather are fixed to some other known constant.

Taken together, our results underscore the importance of conducting additional investigations and devising robust defenses against loop-abort attacks on Fiat–Shamir lattice-based signature schemes.

Acknowledgments

The work described in this paper has been supported by the Einstein Research Unit "Perspectives of a quantum digital transformation: Near-term quantum computational devices and quantum processors" of the Berlin University Alliance. The authors acknowledge the financial support by the Federal Ministry of Education and Research of Germany in the programme of “Souverän. Digital. Vernetzt.” Joint project 6G-RIC, project identification number: 16KISK030 and the project Full Lifecycle Post-Quantum PKI - FLOQI (ID 16KIS1074). We would like to thank Nils Wisiol for his valuable input which greatly improved the paper.

Appendix

Listing 3 The `polyvecl_shuffle` function which shuffles the coefficients in $\mathbf{y}$ using Fisher Yates shuffle [Knu97]. The function `rej_uint16` samples from randomness buf, requesting more randomness from the XOF if required.

```
void polyvecl_shuffle(polyvecl *v, const uint8_t seed[CRHBYTES],
uint16_t nonce) {
  unsigned int current_poly_index, current_coeff_index, i = 0;
  uint8_t random_poly_index, random_coeff_index;
  uint16_t random_flattened_index;
  int32_t tmp_random, tmp_current;
  uint8_t buf[STREAM256_BLOCKBYTES];
  stream256_state state;
  stream256_init(&state, seed, nonce);
  stream256_squeezeblocks(buf, 1, &state);
  for(current_poly_index = ell; current_poly_index-- > 0; ) {
  //ell - 1,..., 0
    for(current_coeff_index = N; current_coeff_index-- > 0; ) {
    //N - 1,..., 0
      // rej_uint16 samples a random number between 0 and current_poly_index * N
      // + current_coeff_index.
      random_flattened_index = rej_uint16(buf, &i, &state,
                                          current_poly_index * N +
                                          current_coeff_index);
      random_poly_index = random_flattened_index / N;
      random_coeff_index = random_flattened_index % N;
      // swap
      tmp_random = v->vec[random_poly_index].coeffs[random_coeff_index];
      tmp_current = v->vec[current_poly_index].coeffs[current_coeff_index];
      v->vec[random_poly_index].coeffs[random_coeff_index] = tmp_current;
      v->vec[current_poly_index].coeffs[current_coeff_index] = tmp_random;
    }
  }
}
```

References

[AAB+19] Frank Arute, Kunal Arya, Ryan Babbush, Dave Bacon, Joseph C Bardin, Rami Barends, Rupak Biswas, Sergio Boixo, Fernando GSL Brandao, David A Buell, et al. Quantum supremacy using a programmable superconducting processor. *Nature*, 574(7779):505–510, 2019.

[AKKM22] Thomas Aulbach, Tobias Kovats, Juliane Krämer, and Soundes Marzougui. Recovering rainbow's secret key with a first-order fault attack. In Lejla Batina and Joan Daemen, editors, *Progress in Cryptology - AFRICACRYPT 2022*, pages 348–368, Cham, 2022. Springer Nature Switzerland.

[AMR+20] Anita Aghaie, Amir Moradi, Shahram Rasoolzadeh, Aein Rezaei Shahmirzadi, Falk Schellenberg, and Tobias Schneider. Impeccable circuits. *IEEE Transactions on Computers*, 69(3):361–376, 2020.

[BBK16] Nina Bindel, Johannes Buchmann, and Juliane Krämer. Lattice-based signature schemes and their sensitivity to fault attacks. In *2016 Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*, pages 63–77. IEEE, 2016.

[BDE+18] Jonathan Bootle, Claire Delaplace, Thomas Espitau, Pierre-Alain Fouque, and Mehdi Tibouchi. LWE without modular reduction and improved side-channel

attacks against BLISS. In Thomas Peyrin and Steven Galbraith, editors, *Advances in Cryptology – ASIACRYPT 2018, Part I*, volume 11272 of *Lecture Notes in Computer Science*, pages 494–524, Brisbane, Queensland, Australia, December 2–6, 2018. Springer, Heidelberg, Germany.

[BDK+] Shi Bai, Leo Ducas, Eike Kiltz, Tancrede Lepoint, Vadim Lyubashevsky, Peter Schwabe, Gregor Seiler, and Damien Stehle. Dilithium reference implementation. https://github.com/pq-crystals/dilithium. [Online; accessed 22-Jun-2023].

[BDK+21] Shi Bai, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, Peter Schwabe, Gregor Seiler, and Damien Stehlé. CRYSTALS-dilithium, algorithm specifications and supporting documentation, 2021. https://pq-crystals.org/dilithium/data/dilithium-specification-round3-20210208.pdf.

[BDSG+14] Johannes Blömer, Ricardo Gomes Da Silva, Peter Günther, Juliane Krämer, and Jean-Pierre Seifert. A practical second-order fault attack against a real-world pairing implementation. In *2014 Workshop on Fault Diagnosis and Tolerance in Cryptography*, pages 123–136. IEEE, 2014.

[BP18] Leon Groot Bruinderink and Peter Pessl. Differential fault attacks on deterministic lattice signatures. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2018(3):21–43, 2018. https://tches.iacr.org/index.php/TCHES/article/view/7267.

[Cen20] NIST Information Technology Laboratory Computer Security Resource Center. NIST Standardization round. https://csrc.nist.gov/projects/post-quantum-cryptography, 2020. [Online; accessed 22-Jun-2023].

[DCG21] Oliver Dial, Jerry Chow, and Jay Gambetta. IBM quantum breaks the 100-qubit processor barrier, Nov 2021. https://research.ibm.com/blog/127-qubit-quantum-processor-eagle.

[Del21] Jeroen Delvaux. Roulette: A diverse family of feasible fault attacks on masked kyber. Cryptology ePrint Archive, Paper 2021/1622, 2021. https://eprint.iacr.org/2021/1622.

[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 Roberto Avanzi and Howard M. Heys, editors, *SAC 2016: 23rd Annual International Workshop on Selected Areas in Cryptography*, volume 10532 of *Lecture Notes in Computer Science*, pages 140–158, St. John's, NL, Canada, August 10–12, 2016. Springer, Heidelberg, Germany.

[FS87] Amos Fiat and Adi Shamir. How to prove yourself: Practical solutions to identification and signature problems. In *Proceedings on Advances in Cryptology—CRYPTO '86*, page 186–194, Berlin, Heidelberg, 1987. Springer-Verlag.

[GBP18] Leon Groot Bruinderink and Peter Pessl. Differential fault attacks on deterministic lattice signatures. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2018(3):21–43, Aug. 2018.

[GJCJ22] Naina Gupta, Arpan Jati, Anupam Chattopadhyay, and Gautam Jha. Lightweight hardware accelerator for post-quantum digital signature crystals-dilithium. Cryptology ePrint Archive, Paper 2022/496, 2022. https://eprint.iacr.org/2022/496.

[GKPM18] Aymeric Genêt, Matthias J. Kannwischer, Hervé Pelletier, and Andrew McLauchlan. Practical fault injection attacks on sphincs. *IACR Cryptol. ePrint Arch.*, 2018:674, 2018.

[Gri19] Roger A Grimes. *Cryptography Apocalypse: Preparing for the Day When Quantum Computing Breaks Today's Crypto*. John Wiley & Sons, 2019.

[Gur23] Gurobi Optimization, LLC. Gurobi Optimizer Reference Manual. https://www.gurobi.com, 2023.

[GW17] Alexandre Gélin and Benjamin Wesolowski. Loop-abort faults on supersingular isogeny cryptosystems. In Tanja Lange and Tsuyoshi Takagi, editors, *Post-Quantum Cryptography - 8th International Workshop, PQCrypto 2017*, pages 93–106, Utrecht, The Netherlands, June 26–28, 2017. Springer, Heidelberg, Germany.

[HTS11] Yasufumi Hashimoto, Tsuyoshi Takagi, and Kouichi Sakurai. General fault attacks on multivariate public key cryptosystems. In Bo-Yin Yang, editor, *Post-Quantum Cryptography*, pages 1–18, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[Int19] Intel Corporation. Intel introduces 'horse ridge' to enable commercially viable quantum computers. https://newsroom.intel.de/news/intel-introduces-horse-ridge-to-enable-commercially-viable-quantum-computers/, Dec 2019.

[Knu97] Donald Knuth. *The Art of Computer Programming*, volume 2. Addison-Wesley, 3 edition, 1997.

[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.

[LSG22] Georg Land, Pascal Sasdrich, and Tim Güneysu. A hard crystal - implementing dilithium on reconfigurable hardware. In Vincent Grosso and Thomas Pöppelmann, editors, *Smart Card Research and Advanced Applications*, pages 210–230, Cham, 2022. Springer International Publishing.

[Lyu09] Vadim Lyubashevsky. Fiat-Shamir with aborts: Applications to lattice and factoring-based signatures. In Mitsuru Matsui, editor, *Advances in Cryptology – ASIACRYPT 2009*, volume 5912 of *Lecture Notes in Computer Science*, pages 598–616, Tokyo, Japan, December 6–10, 2009. Springer, Heidelberg, Germany.

[MGTF19] Vincent Migliore, Benoît Gérard, Mehdi Tibouchi, and Pierre-Alain Fouque. Masking dilithium. In Robert H. Deng, Valérie Gauthier-Umaña, Martín Ochoa, and Moti Yung, editors, *Applied Cryptography and Network Security*, pages 344–362, Cham, 2019. Springer International Publishing.

[MHS+19] Sarah McCarthy, James Howe, Neil Smyth, Séamus Brannigan, and Máire O'Neill. Bearz attack falcon: Implementation attacks with countermeasures on the falcon signature scheme. In *IACR Cryptology ePrint Archive*, 2019.

[Mos18] Michele Mosca. Cybersecurity in an era with quantum computers: Will we be ready? *IEEE Security & Privacy*, 16:38–41, 09 2018.

[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. *Cryptology ePrint Archive*, 2022.

[OC14] Colin O'Flynn and Zhizhang (David) Chen. ChipWhisperer: An open-source platform for hardware embedded security research. In Emmanuel Prouff, editor, *COSADE 2014: 5th International Workshop on Constructive Side-Channel Analysis and Secure Design*, volume 8622 of *Lecture Notes in Computer Science*, pages 243–260, Paris, France, April 13–15, 2014. Springer, Heidelberg, Germany.

[Pes16] Peter Pessl. Analyzing the shuffling side-channel countermeasure for lattice-based signatures. In Orr Dunkelman and Somitra Kumar Sanadhya, editors, *Progress in Cryptology - INDOCRYPT 2016: 17th International Conference in Cryptology in India*, volume 10095 of *Lecture Notes in Computer Science*, pages 153–170, Kolkata, India, December 11–14, 2016. Springer, Heidelberg, Germany.

[PSS+18] Damian Poddebniak, Juraj Somorovsky, Sebastian Schinzel, Manfred Lochter, and Paul Rösler. Attacking deterministic signature schemes using fault attacks. In *2018 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 338–352, 2018.

[PV06] Dan Page and Frederik Vercauteren. A fault attack on pairing-based cryptography. *IEEE Transactions on Computers*, 55(9):1075–1080, 2006.

[Rig21] Rigetti Computing. Rigetti computing announces next-generation 40q and 80q quantum systems, Dec 2021.

[RJH+18] Prasanna Ravi, Mahabir Prasad Jhanwar, James Howe, Anupam Chattopadhyay, and Shivam Bhasin. Side-channel assisted existential forgery attack on Dilithium - A NIST PQC candidate. Cryptology ePrint Archive, Report 2018/821, 2018. https://eprint.iacr.org/2018/821.

[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 *Proceedings of the 2019 ACM Asia Conference on Computer and Communications Security*, Asia CCS '19, page 427–440, New York, NY, USA, 2019. Association for Computing Machinery.

[SBB+18] Niels Samwel, Lejla Batina, Guido Bertoni, Joan Daemen, and Ruggero Susella. Breaking ed25519 in wolfssl. In Nigel P. Smart, editor, *Topics in Cryptology – CT-RSA 2018*, pages 1–20, Cham, 2018. Springer International Publishing.

[YW06] Chih-Hsu Yen and Bing-Fei Wu. Simple error detection methods for hardware implementation of advanced encryption standard. *IEEE Transactions on Computers*, 55(6):720–731, 2006.

[ZHL+21] Zhen Zhou, Debiao He, Zhe Liu, Min Luo, and Kim-Kwang Raymond Choo. A software/hardware co-design of crystals-dilithium signature scheme. *ACM Trans. Reconfigurable Technol. Syst.*, 14(2), jun 2021.

[ZZW+21] Cankun Zhao, Neng Zhang, Hanning Wang, Bohan Yang, Wenping Zhu, Zhengdong Li, Min Zhu, Shouyi Yin, Shaojun Wei, and Leibo Liu. A compact and

high-performance hardware architecture for crystals-dilithium. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2022(1):270–295, Nov. 2021.