IACR Transactions on Cryptographic Hardware and Embedded Systems
ISSN 2569-2925, Vol. 2023, No. 2, pp. 447–481. DOI:10.46586/tches.v2023.i2.447-481

Fiddling the Twiddle Constants - Fault Injection Analysis of the Number Theoretic Transform

Prasanna Ravi[1,2†], Bolin Yang[3,4†], Shivam Bhasin[1], Fan Zhang[3,4,5,6] and Anupam Chattopadhyay[1,2]

[1] Temasek Laboratories, Nanyang Technological University, Singapore
[2] School of Computer Science and Engineering, Nanyang Technological University, Singapore
[3] Zhejiang University, Hangzhou, China
[4] Alibaba-Zhejiang University Joint Institute of Frontier Technologies, Hangzhou, China
[5] ZJU-Hangzhou Global Scientific and Technological Innovation Center, Hangzhou, China
[6] Jiaxing Research Institute, Zhejiang University, Jiaxing, China
prasanna.ravi@ntu.edu.sg yangbolin@zju.edu.cn sbhasin@ntu.edu.sg
fanzhang@zju.edu.cn anupam@ntu.edu.sg

Abstract.
In this work, we present the first fault injection analysis of the Number Theoretic Transform (NTT). The NTT is an integral computation unit, widely used for polynomial multiplication in several structured lattice-based key encapsulation mechanisms (KEMs) and digital signature schemes. We identify a critical single fault vulnerability in the NTT, which severely reduces the entropy of its output. This in turn enables us to perform a wide-range of attacks applicable to lattice-based KEMs as well as signature schemes. In particular, we demonstrate novel key recovery and message recovery attacks targeting the key generation and encryption procedure of Kyber KEM. We also propose novel existential forgery attacks targeting deterministic and probabilistic signing procedure of Dilithium, followed by a novel verification bypass attack targeting its verification procedure. All proposed exploits are demonstrated with high success rate using electromagnetic fault injection on optimized implementations of Kyber and Dilithium, from the open-source *pqm4* library on the ARM Cortex-M4 microcontroller. We also demonstrate that our proposed attacks are capable of bypassing concrete countermeasures against existing fault attacks on lattice-based KEMs and signature schemes. We believe our work motivates the need for more research towards development of countermeasures for the NTT against fault injection attacks.

Keywords: Lattice-based cryptography · Electromagnetic Fault-Injection attack · Number Theoretic Transform · Learning With Error · Kyber · Dilithium

1 Introduction

The NIST standardization process for *post-quantum cryptography* has finished its third round, and provided a list of new public key schemes for new standardization [AAC+22]. While implementation performance and theoretical security guarantees served as the main criteria in the initial rounds, resistance against *side-channel attacks* (SCA) and *fault injection attacks* (FIA) emerged as an important criterion in the final round, as also clearly stated by NIST at several instances [AH21, RR21].

†The first two authors contributed equally to this work.

Received: 2022-10-15 Accepted: 2022-12-15 Published: 2023-03-06

Amongst the seven main finalists in third round for *key encapsulation mechanisms* (KEMs) and *digital signatures*, five schemes base their security on hard problems over structured lattices [AASA+20]. These schemes are particularly attractive for constrained embedded devices, owing to their relatively small public key sizes and highly competitive runtimes. They typically operate over polynomials in polynomial rings, and notably, *polynomial multiplication* is one of the most computationally intensive operations in practical implementations of these schemes. Among the several known techniques for polynomial multiplication such as the schoolbook multiplier, Toom-Cook [Too63] and Karatsuba [Kar63], the Number Theoretic Transform (NTT) based polynomial multiplication [CT65] is one of the most widely adopted techniques, owing to its superior run-time complexity and a compact design. Over the years, there has been a sustained effort by the cryptographic community to improve the performance of NTT for lattice-based schemes on a wide-range of hardware and software platforms [RVM+14, POG15, BKS19, ACC+22, CHK+21]. As a result, the use of NTT for polynomial multiplication yields the fastest implementation for several lattice-based schemes. In particular, the NTT serves as a critical computational kernel used in Kyber [ABD+20] and Dilithium [LDK+17], which were selected as the first candidates for PQC standardization.

While the NTT provides significant implementation benefits, it also manipulates sensitive variables, thereby serving as an attractive target for SCA and FIA. While the side-channel resistance of the NTT has been studied by a number of works [PPM17a, PP19, RPBC20], its susceptibility to fault injection attacks has not received much attention. Given its widespread use in lattice-based schemes, this raises a critical question whether *the NTT or more importantly its implementations contain hidden vulnerabilities that can be exploited through FIA to compromise the security of lattice-based schemes.*

Our Contribution: In this work, we answer this question positively, by presenting the first *fault injection analysis* of the NTT. Our work relies on a key observation that *zeroization of the twiddle constants* significantly reduces the entropy in the NTT output, which in turn severely impacts the security of lattice-based schemes. To analyze the feasibility of such a fault, we perform a detailed study of the optimized implementations of the NTT used in Kyber (representative of KEMs) and Dilithium (representative of signature schemes) on the ARM Cortex-M4 microcontroller using electromagnetic fault injection. We identified a critical fault vulnerability in their implementations, which enables zeroization of all the twiddle constants using a *single targeted fault*. This enables practical key/message recovery attacks on Kyber KEM and forgery attacks on Dilithium. The proposed attacks are also shown to bypass most known fault countermeasures for lattice-based KEMs and signature schemes. To the best of our knowledge, we present first practical forgery attack on the probabilistic variant[1] and *verification bypass* attack on the verification procedure of Dilithium.

Organization of the Paper

In Section 2, we provide a generic description of Kyber and Dilithium, and provide some background about the NTT as well as related prior work. In Section 3, we show related works about FA on Lattice-based cryptography and classify them. In Section 4, we describe the identified vulnerability in the NTT, and a detailed analysis of the same over practical implementations of the NTT in Kyber and Dilithium. In Sections 5 and 6, we demonstrate exploitation of the identified vulnerability in Kyber and Dilithium respectively. In Section 7, we perform experimental validation of our attacks using EMFI on unprotected and protected targets, followed by conclusion and mitigation in Section 8.

[1] Islam et al. [IMS+22] recently proposed a rowhammer based attack on deterministic and probabilistic Dilithium but its final complexity still remains as 2^{89}, while we report a full break.

2 Background

2.1 Notation

Let q be a prime number, and the field of integers modulo q be denoted as $\mathbb{Z}_q$. Schemes such as Kyber and Dilithium operate over polynomials in polynomial rings. The polynomial ring $\mathbb{Z}_q[x]/\phi(x)$ is denoted as R_q where $\phi(x) = x^n + 1$ is a cyclotomic polynomial with n being a power of 2. Polynomials in R_q are denoted using regular font letters (i.e.) $a \in R_q$. The i^{th} coefficient of $a \in R_q$ is denoted as $a_i \in \mathbb{Z}_q$. For $a \in R_q$, $\ell_\infty(a)$ denotes the largest absolute value of a coefficient of a in $\mathbb{Z}_q$. A vector of polynomials in R_q is denoted using bold lower case letters (i.e.) $\mathbf{a} \in R_q^k$ with $k > 1$, and a matrix of polynomials in R_q is denoted using bold upper case letters (i.e.) $\mathbf{A} \in R_q^{k \times \ell}$ with $(k, \ell) > 1$. The element $\mathbf{A}[i][j]$ denotes the polynomial in row i and j of $\mathbf{A} \in R_q^{k \times \ell}$. Transpose of a matrix $\mathbf{A}$ is denoted as $\mathbf{A}^T$. Multiplication of polynomials $a, b \in R_q$ is denoted as $c = a \cdot b \in R_q$. Pointwise/Coefficient-wise multiplication of two polynomials $a, b \in R_q$ is denoted as $c = a \circ b \in R_q$, which means that each of the coefficients of polynomial a multiplies the coefficients of b with the same index. We denote $\mathcal{B}$ as a byte array, where the i^{th} byte is denoted as $\mathcal{B}[i]$. A bit-string is denoted using regular lower case font letters (i.e.) $m \in \{0,1\}^*$. For a given element a ($\mathbb{Z}_q$ or R_q or $R_q^{k \times \ell}$), its corresponding faulty value is denoted as a^* and we utilize this notation for description of our attacks. The NTT representation of a polynomial $a \in R_q$ is denoted as $\hat{a} \in R_q$, and the same notation also applies to modules of higher dimension.

2.2 Number Theoretic Transform

The Number Theoretic Transform (NTT) is utilized as a building block for polynomial multiplication operation in several structured lattice-based schemes. While schemes such as Kyber and Dilithium were designed with *NTT-friendly* parameters to allow use of NTT, other schemes such as Saber, NTRU and NTRU Prime were designed with *NTT-unfriendly* parameters, thereby relying on other techniques such as Toom-Cook [Coo66] and Karatsuba [Kar63] for polynomial multiplication. However, recent works such as [ACC+21, CHK+21, ACC+22] have shown that NTT can be indeed be used in these schemes, which also leads to significant improvement in performance over non-NTT based approaches.

The NTT is simply a bijective mapping for a polynomial $p \in R_q$ from a *normal* domain into an alternative represetation $\hat{p} \in R_q$ in the *NTT domain* as follows:

$$\hat{p}_j = \sum_{i=0}^{n-1} p_i \cdot \omega^{i \cdot j} \tag{1}$$

where $j \in [0, n-1]$ and ω is the n^{th} root of unity in the operating ring $\mathbb{Z}_q$.

The corresponding inverse operation named Inverse NTT (denoted as INTT) maps $\hat{p}$ in the NTT domain back to p in the normal domain as follows:

$$p_j = \frac{1}{n} \sum_{i=0}^{n-1} \hat{p}_i \cdot \omega^{-i \cdot j} \tag{2}$$

The use of NTT requires either the n^{th} root of unity (ω) or $2n^{\text{th}}$ root of unity (ψ) in the underlying ring $\mathbb{Z}_q$ ($\psi^2 = \omega$), which can be ensured through appropriate choices for the parameters (n, q). The powers of ω and ψ that are used within the NTT computation are commonly referred to as *twiddle constants*. NTT based multiplication of two polynomials $\mathbf{a}$ and $\mathbf{b}$ in R_q is typically done as follows:

$$\mathbf{c} = \text{INTT}(\text{NTT}(\mathbf{a}) \circ \text{NTT}(\mathbf{b})). \tag{3}$$

The NTT over an n point sequence is performed using the well-known *butterfly* network, which operates over several layers/stages. The atomic operation within the NTT computation is denoted as the *butterfly* operation. A butterfly operation takes as inputs $(a, b) \in \mathbb{Z}_q^2$ and a twiddle constant w, and produces outputs $(c, d) \in \mathbb{Z}_q^2$. There are two types of butterfly operations, which can be interchangeably used in the NTT/INTT: (1) Cooley-Tukey (CT) butterfly [CT65] in Eqn.4 and (2) Gentleman-Sande (GS) butterfly [GS66] in Eqn.5.

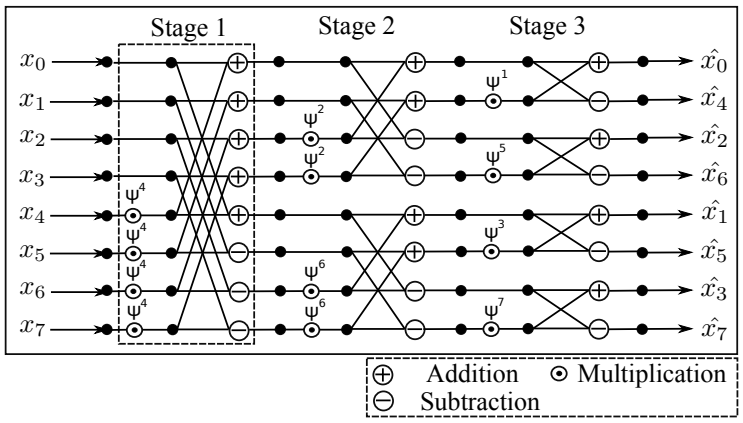


Figure 1: Data flow graphs of CT-butterfly based NTT for size $n = 8$.

An NTT/INTT of size $n = 2^k$ typically consists of k stages with each stage containing $n/2$ butterfly operations. We refer to Fig.1 for the data-flow graph of a CT-butterfly based NTT for an input sequence with length $n = 8$.

$$\begin{aligned} c &= a + b \cdot w \\ d &= a - b \cdot w, \end{aligned} \tag{4}$$

$$\begin{aligned} c &= a + b \\ d &= (a - b) \cdot w, \end{aligned} \tag{5}$$

The underlying integer ring $\mathbb{Z}_q$ of Dilithium contains both ω and ψ, ensuring complete factorization of $(x^n + 1)$ into linear factors (degree 1). This enables to use a *complete* NTT with $k = \log_2(n)$ stages. However, the ring $\mathbb{Z}_q$ of Kyber only contains ω, which implies that $(x^n + 1)$ can only be factored into $n/2$ quadratic factors (degree 2). Thus, the last stage of NTT/INTT in Kyber is skipped and the NTT output contains $n/2$ elements. Thus, Kyber relies on the use of an *incomplete* NTT with $k - 1$ stages.

2.3 Kyber

Kyber is a Chosen-Ciphertext Attack (CCA) secure KEM based on the Module Learning With Errors (M-LWE) problem. Computations are done over modules in dimension $(k \times k)$ (i.e) $R_q^{k \times k}$ where $R_q = \mathbb{Z}_q[x]/(x^n + 1)$, $q = 3329$ and $n = 256$. Kyber comes in three security levels, Kyber512 (NIST Level 1), Kyber-768 (Level 3) and Kyber-1024 (Level 5) with $k = 2, 3$ and 4 respectively. The parameters q, n and the modular polynomial $\phi(x) = x^n + 1$ are chosen, so as to allow the use of the Number Theoretic Transform (NTT) for polynomial multiplication in R_q.

The CCA secure Kyber KEM contains in its core, a Chosen-Plaintext Attack (CPA) secure PKE. We refer to Algorithm 1 for a simplified description of the key-generation and encryption procedures of CPA secure PKE of Kyber. We do not describe the decryption procedure, as it is not a target of our attacks. The function Sample_U samples from a uniform distribution, Sample_B samples from a binomial distribution; Expand expands a

small seed into a uniformly random matrix in $R_q^{k\times k}$. The function $\mathsf{Compress}(u,d)$ lossily compresses $u \in \mathbb{Z}_q$ into $v \in \mathbb{Z}_{2^d}$ with $q > 2^d$, while $\mathsf{Decompress}(v,d)$ extrapolates $v \in \mathbb{Z}_{2^d}$ into $u' \in \mathbb{Z}_q$. Both Compress and Decompress can also be applied over vectors, where the function is simply computed in a component-wise fashion.

The CPA secure PKE is converted into a CCA secure KEM using the Fujisaki-Okamoto transformation [FO99]. The CPA.Encrypt (resp. CPA.Decrypt) procedure is converted into the encapsulation procedure CCA.Encaps procedure (resp. decapsulation procedure CPA.Decaps). The encapsulation procedure (CCA.Encaps) uses the public key pk and instantiates the CPA.Encrypt procedure to generate the ciphertext ct for an internally generated message m. It also generates a corresponding session key K.

The decapsulation procedure (CPA.Decaps) uses the secret key sk to decrypt the ciphertext ct into the message m, and re-encrypts the message to compute a new ciphertext $\bar{ct}$. Subsequently, ct is compared with $\bar{ct}$, and if the comparison succeeds, a valid session key K is generated. Otherwise, the ciphertext ct is considered invalid, and a random session key K is generated. This enables to detect invalid ciphertexts, thereby offering concrete theoretical security guarantees against chosen-ciphertext attacks. We refer the reader to [ABD+20] for more details on CCA secure Kyber KEM.

Algorithm 1 CPA Secure Kyber PKE (Simplified)

procedure CPA.KeyGen
  $seed_A \in \mathcal{B} \leftarrow \mathsf{Sample}_U()$ ▷ Generate uniform $Seed_A$
  $seed_B \in \mathcal{B} \leftarrow \mathsf{Sample}_U()$ ▷ Generate uniform $Seed_B$
  $\hat{\mathbf{A}} = \mathsf{NTT}(\mathbf{A}) \in R_q^{k\times k} \leftarrow \mathsf{Expand}(seed_A)$ ▷ Expand $seed_A$ into $\hat{\mathbf{A}}$ in NTT domain
  $\mathbf{s} \in R_q^k \leftarrow \mathsf{Sample}_B(seed_B, coins_s)$ ▷ Sample secret $\mathbf{s}$ using $(Seed_B, coins_s)$
  $\mathbf{e} \in R_q^k \leftarrow \mathsf{Sample}_B(seed_B, coins_e)$ ▷ Sample error $\mathbf{e}$ using $(Seed_B, coins_e)$
  $\hat{\mathbf{s}} \in R_q^k \leftarrow \mathsf{NTT}(\mathbf{s})$ ▷ $\mathsf{NTT}(\mathbf{s})$
  $\hat{\mathbf{e}} \in R_q^k \leftarrow \mathsf{NTT}(\mathbf{e})$ ▷ $\mathsf{NTT}(\mathbf{e})$
  $\hat{\mathbf{t}} = \hat{\mathbf{A}} \circ \hat{\mathbf{s}} + \hat{\mathbf{e}}$ ▷ $\mathbf{t} = \mathbf{A}\cdot\mathbf{s} + \mathbf{e}$ in NTT domain
  Return $(pk = (seed_A, \hat{\mathbf{t}}), sk = (\hat{\mathbf{s}}))$
end procedure

procedure CPA.Encrypt$(pk, m \in \{0,1\}^{256}, seed_R \in \{0,1\}^{256})$
  $\hat{\mathbf{A}} \in R_q^{k\times k} \leftarrow \mathsf{Expand}(seed_A)$
  $\mathbf{r} \in R_q^k \leftarrow \mathsf{Sample}_B(seed_R, coins_0)$ ▷ Sample $\mathbf{r}$ using $(Seed_R, coins_0)$
  $\mathbf{e_1} \in R_q^k \leftarrow \mathsf{Sample}_B(seed_R, coins_1)$ ▷ Sample $\mathbf{e}_1$ using $(Seed_R, coins_1)$
  $\mathbf{e_2} \in R_q^k \leftarrow \mathsf{Sample}_B(seed_R, coins_2)$ ▷ Sample $\mathbf{e}_2$ using $(Seed_R, coins_2)$
  $\hat{\mathbf{r}} \in R_q^k \leftarrow \mathsf{NTT}(\mathbf{r})$ ▷ $\mathsf{NTT}(\mathbf{r})$
  $\mathbf{u} \in R_q^k \leftarrow \mathsf{INTT}(\mathbf{A}^T \circ \hat{\mathbf{r}}) + \mathbf{e_1}$ ▷ $\mathbf{u} = \mathbf{A}^T\cdot\mathbf{r} + \mathbf{e_1}$
  $v \in R_q \leftarrow \mathsf{INTT}(\hat{t}^T \circ \hat{\mathbf{r}}) + \mathbf{e_2} + \mathsf{Decompress}(m,1)$ ▷ $\mathbf{v} = \mathbf{t}^T\cdot\mathbf{r} + \mathbf{e_2} + \mathsf{Encode}(m)$
  Return $ct = \mathsf{Compress}(\mathbf{u}, d_1), \mathsf{Compress}(v, d_2)$
end procedure

2.4 Dilithium

Dilithium is a lattice-based digital signature scheme, whose security is based on the Module LWE (M-LWE) and Module SIS (M-SIS) problem. Dilithium operates over the module $R_q^{k\times\ell}$ with $(k,\ell) > 1$ where $R_q = \mathbb{Z}[x]/(x^n+1)$, $n = 256$ and $q = 2^{23} - 2^{13} - 1$. This choice of parameters allows the use of the NTT for polynomial multiplication in R_q. Dilithium also comes in three security levels: Dilithium2 with $(k,\ell) = (4,4)$ at NIST Level 2, Dilithium3 with $(k,\ell) = (6,5)$ at NIST Level 3 and Dilithium5 with $(k,\ell) = (8,7)$ at NIST Level

Algorithm 2 Dilithium Signature scheme (Simplified)

procedure KeyGen
  $(seed_A, seed_S, K) \in \mathcal{B} \leftarrow \mathsf{Sample}_U()$; $\mathbf{s_1}, \mathbf{s_2} \in (R_q^\ell \times R_q^k) \leftarrow \mathsf{Sample}_B(seed_S)$
  $\mathbf{A} \in R_q^{k \times \ell} \leftarrow \mathsf{Expand}(seed_A)$
  $\mathbf{t} = \mathbf{A} \cdot \mathbf{s_1} + \mathbf{s_2}$ ▷ Generate LWE instance $\mathbf{t}$
  $(\mathbf{t_1}, \mathbf{t_0}) \leftarrow \mathsf{Power2Round}(\mathbf{t})$ ▷ Split $\mathbf{t}$ as $\mathbf{t_1} \cdot 2^d + \mathbf{t_0}$
  $tr \in \mathcal{B} \leftarrow \mathcal{H}(seed_A \| \mathbf{t}_1)$
  $pk = (seed_A, \mathbf{t_1}), sk = (seed_A, K, tr, \mathbf{s_1}, \mathbf{s_2}, \mathbf{t_0})$
end procedure

---

procedure Sign(sk, M)
  $\hat{\mathbf{A}} \in R_q^{k \times \ell} \leftarrow \mathsf{Expand}(seed_A)$
  $\mu \in \{0,1\}^{512} \leftarrow \mathcal{H}(tr \| M)$ ▷ Hash m with public value tr
  $\kappa \leftarrow 0$; $(\mathbf{z}, \mathbf{h}) \leftarrow \perp$
  if Deterministic **then**
    $\rho \in R_q^\ell \leftarrow \mathcal{H}(K \| \mu)$ ▷ Generate seed ρ using message and secret seed K
  else
    $\rho \in R_q^\ell \leftarrow \mathsf{Sample}_U()$ ▷ Generate uniform seed ρ
  end if
  while $(\mathbf{z}, \mathbf{h}) = \perp$ **do** ▷ Start of Abort Loop
    $\mathbf{y} \leftarrow \mathsf{Sample}_Y(\rho \| \kappa)$
    $\hat{\mathbf{y}} = \mathsf{NTT}(\mathbf{y})$ ▷ $\mathsf{NTT}(y)$
    $\mathbf{w} \leftarrow \mathsf{INTT}(\hat{\mathbf{A}} \circ \hat{\mathbf{y}})$; $\mathbf{w_1} \leftarrow \mathsf{HighBits}(\mathbf{w})$ ▷ $\mathsf{w}_1 = \mathsf{HighBits}(\mathbf{A} \cdot \mathbf{y})$
    $c \in R_q \leftarrow \mathcal{H}(\mu \| \mathbf{w_1})$ ▷ Generate Sparse Challenge c
    $\hat{c} = \mathsf{NTT}(c)$ ▷ $\mathsf{NTT}(c)$
    $\mathbf{z} = \mathsf{INTT}(\hat{\mathbf{c}} \circ \hat{\mathbf{s}}_\mathbf{1}) + \mathbf{y}$ ▷ $\mathbf{z} = \mathbf{s_1} \cdot c + \mathbf{y}$
    ...
    Compute Hint Vector $\mathbf{h}$
    if Conditional Checks Not Satisfied **then**
      $(\mathbf{z}, \mathbf{h}) = \perp$
      $\kappa = \kappa + 1$
    end if
  end while
  $\sigma = (\mathbf{z}, \mathbf{h}, c)$
end procedure

---

procedure Verify($pk, M, \sigma = (\mathbf{z}, \mathbf{h}, c)$)
  $\mu \in \{0,1\}^{512} \leftarrow \mathcal{H}(tr \| M)$
  $\hat{c} = \mathsf{NTT}(c)$ ▷ $\mathsf{NTT}(c)$
  $\mathbf{w}'_1 := \mathsf{UseHint}(\mathbf{h}, \mathbf{A} \cdot \mathbf{z} - \mathsf{INTT}(\hat{c} \circ \hat{\mathbf{t}}_\mathbf{1} \cdot 2^d, 2\gamma_2)$
  $\bar{c} = \mathcal{H}(\mu, \mathbf{w}'_1)$
  if $(\bar{c} == c)$ and (norm of $\mathbf{z}$ and $\mathbf{h}$ are valid) **then**
    Return Pass
  else
    Return Fail
  end if
end procedure

5. There are two variants of Dilithium: (1) Deterministic (2) Probabilistic/Randomized, which only subtly differ in the way randomness is used in the signing procedure. The signing procedure of the deterministic Dilithium does not utilize external randomness and can generate only a single signature for a given message. The randomized variant however

utilizes external randomness and thus generates a different signature, for a given message in each execution.

Refer Alg.2 for the key generation, signing and verification procedures of Dilithium. The functions Sample_U, Sample_B and Expand perform the same functions as in Kyber, albeit with different parameters. Dilithium also uses a number of rounding functions such as $\mathsf{Power2Round}$, $\mathsf{HighBits}$, $\mathsf{LowBits}$, $\mathsf{MakeHint}$ and $\mathsf{UseHint}$, whose details can be found in [LDK+17]. The key generation procedure simply involves generation of an LWE instance $\mathbf{t}$ (Line 4). Subsequently, the LWE instance is split into higher and lower order bits $\mathbf{t_1}$ and $\mathbf{t_0}$ respectively (Line 5), where $\mathbf{t_1}$ forms part of the public key, while $\mathbf{t_0}$ becomes part of the secret key.

The signing procedure of Dilithium is based on the "Fiat-Shamir with Aborts" framework where the signature is repeatedly generated and rejected until it satisfies a given set of conditions [Lyu09]. The message m is first hashed with a public value tr to generate μ (Line 11). The abort loop (Line 18-31) starts by generating an ephemeral nonce $\mathbf{y} \in R_q^\ell$, using a seed ρ. For the deterministic variant, the seed ρ is obtained by hashing μ with a secret nonce K (Line 14), while the probabilistic variant randomly samples the seed ρ from a uniform distribution (Line 16). This is the only differentiator between the two variants. The nonce $\mathbf{y}$ along with the public key component $\mathbf{A}$ is then used to calculate a sparse challenge polynomial $c \in R_q$ (Line 22), whose 60 coefficients are either ± 1, while the other 196 coefficients are 0. Subsequently, the challenge c, nonce $\mathbf{y}$ and secret $\mathbf{s}_1$, are used to compute the primary signature component $\mathbf{z}$ (Line 24). Then, a hint vector $\mathbf{h}$ is generated and output as part of the signature σ. The abort loop contains several conditional checks (Line 27), which should be simultaneously satisfied to terminate the abort loop and generate the signature $\sigma = (\mathbf{z}, \mathbf{h}, c)$.

The verification procedure utilizes the signature σ and the public key pk to recompute the challenge polynomial $\bar{c}$ (Line 38), which is then compared with the received challenge c, along with other checks (Line 39). If all the checks are satisified, then the verification is successful, else it is a failure.

3 Prior Works

In this section, we discuss existing works that explore vulnerability of lattice-based KEMs and digital signature schemes against fault-injection attacks and corresponding countermeasures.

3.1 Fault Attacks on Signature Schemes

With respect to signature schemes, we focus on attacks targeting the signing and verification procedure, while attacks on the key-generation procedure are considered out of scope.

3.1.1 Targeting the Signing Procedure

We categorize attacks on the signing procedure into the following categories, depending on the type of fault models and target operations.

1. **Randomization Faults**
2. **Skipping Faults**
3. **Zeroization Faults**

1.Randomization Faults: The attack involves injection of random faults to either (1) corrupt targeted variables or (2) alter control flow of the signing procedure.

(a) Randomize_Secret_Key Attack: Bindel *et al.* [BBK16] reported the first fault vulnerability analysis of lattice-based signature schemes such as GLP [GLP12] and BLISS [DDLL13], based on the "Fiat-Shamir with Aborts" framework. They proposed to inject random faults to change a single or few coefficients of the secret module $\mathbf{s}_1 \in R_q^\ell$. The attacker can subsequently utilize the knowledge of a few hundred faulty signatures to fully recover $\mathbf{s}_1$. Knowledge of $\mathbf{s}_1$ alone is sufficient for an attacker to forge signatures of Dilithium, as shown in [RJH+19, BP18].

Along the same lines, Islam *et al.* [IMS+22] recently presented a novel signature correction attack, which also works by injecting random bit flips in single coefficients of the secret module $\mathbf{s}_1$, stored in memory. They utilize Rowhammer as an attack vector to inject random bit flips, and subsequently utilized a signature correction algorithm on the faulty signatures to recover the secret key. We henceforth refer to these attacks faulting the secret key as Randomize_Secret_Key fault attacks.

Countermeasure against Randomize_Secret_Key *attack:* The faulty signatures generated due to injection of randomization faults are invalid with an overwhelming probability. Thus, verifying the validity of the generated signatures serves as a concrete countermeasure. The countermeasure is also effective against any future fault attacks which produce invalid signatures. We henceforth refer to this countermeasure as Verify_After_Sign countermeasure.

(b) Generic_DFA Attack: Bruinderink and Pessl [BP18] presented a powerful Differential Fault Attack (DFA), particularly applicable to the deterministic variant of Dilithium, whose modus operandi is as follows: the attacker has access to a signing oracle, and submits a signature query for a randomly chosen message m. Let the primary signature component be $\mathbf{z} = \mathbf{s}_1 \cdot c + \mathbf{y}$. The attacker then submits a signing query for the same message m, but injects a random fault such that the corresponding faulty signature is $\mathbf{z}' = \mathbf{s}_1 \cdot c' + \mathbf{y}$, which is computed with the same nonce $\mathbf{y}$, but with a different challenge polynomial c'. The difference $\Delta\mathbf{z} = \mathbf{z} - \mathbf{z}'$ can be used to trivially recover the entire secret module $\mathbf{s}_1$, with only a single faulty signature. The authors showed that a single random fault anywhere within 68% of the execution time of the signing procedure can result in full key recovery, thereby demonstrating the effectiveness of their attack. We henceforth refer to this attack as the Generic_DFA attack on Dilithium.

Countermeasure against Generic_DFA *attack:* Similar to the Randomize_Secret_Key attack, Generic_DFA attack also results in invalid signatures which do not pass verification. Thus, the Verify_After_Sign countermeasure serves as a strong deterrent against the attack. However, the authors of [BP18] also showed an interesting variant of their attack which works by injecting faults during sampling of $\mathbf{y}$, that results in valid signatures. Thus this variant of their attack can bypass the Verify_After_Sign countermeasure.

2.Skipping Faults: This class of attacks work by injecting faults to skip targeted instructions in the signing procedure.

(a) Loop_Abort Attack: Espitau *et al.* [EFGT16] presented a novel *loop abort* fault attack on the signing procedure of BLISS, to prematurely abort the sampling of the nonce $\mathbf{y}$ (equivalent to Line 19 in Alg.2). This results in generation of $\mathbf{y}$ with very low degree (i.e.) with several zero coefficients. Utilization of such a sparse nonce $\mathbf{y}$ to generate signatures leads to easy recovery of $\mathbf{s}_1$, even with a single such faulty signature [EFGT16]. We refer to this attack as the Loop_Abort fault attack.

Countermeasure against Loop_Abort *fault attack:* The attack works by injecting faults in

the value of $\mathbf{y}$. Thus, the generated faulty signatures are valid. Thus, the attack can easily bypass the Verify_After_Sign countermeasure. However, the attack can be mitigated using implementation level countermeasures such as a loop counter, that keeps track of the number of sampled coefficients of $\mathbf{y}$. While one can argue that the countermeasure can also be bypassed through faults, it is possible to design the countermeasure in a careful manner, so as to avoid such trivial double fault injection attacks. The loop counter can be implemented in the following manner. The number of coefficients of $\mathbf{y} \in R_q^\ell$ is $(\ell \cdot n)$. We sample a random integer $g \in \mathbb{Z}^+$. We initialize a loop counter lc to 0 and its value is increased by g for every sampled coefficient of $\mathbf{y}$. Subsequently, the generated signature σ is stored in a temporary variable *temp*, and is copied one byte at a time to the output variable *sig* (initialized with 0), only if the loop counter value is equal to the expected value $(\ell \cdot n \cdot g)$. This comparison is done for every byte moved from *temp* to *sig*. In essence, the signature is passed onto the output, only if all the coefficients of $\mathbf{y}$ have been sampled.

Such use of a dynamic loop counter whose value changes for every execution ($lc = \ell \cdot n \cdot g$), provides increased resistance against double fault attacks, which target the loop counter protection. Injection of very precise faults to force successful comparison is challenging to achieve in practice. Moreover, simply skipping the loop counter comparison results in a zero signature ($sig = 0$), which is not useful for an attacker. We refer to this countermeasure as the Verify_Loop_Abort countermeasure.

(b) Skip_Addition Attack: Bindel *et al.* [BBK16] proposed theoretical skipping fault attacks targeting the final addition operation used to generate $\mathbf{z} \in R_q^\ell$ (Line 24 in Alg.2). Skipping the addition of $\mathbf{y} \in R_q^\ell$ with the product $(\mathbf{s}_1 \cdot c) \in R_q^\ell$, unmasks the coefficients of the product $(\mathbf{s}_1 \cdot c)$, whose knowledge can be used to recover $\mathbf{s}_1$. While this is possible by skipping the entire addition operation, Ravi *et al.* [RJH+19] proposed a more subtle fault attack on the deterministic variant of Dilithium, which involves skipping of the addition operation for single coefficients of $\mathbf{z}$. An attacker can then use a DFA technique similar to [BP18], to recover the secret module $\mathbf{s}_1$ in only a few hundred such faulty signatures. We refer to these attacks as the Skip_Addition fault attacks.

Countermeasure against Skip_Addition *attack:* The dynamic loop counter protection can be used to keep track of the number of addition operations to generate the primary signature component $\mathbf{z}$. However, the protection does not defeat attacks that skip addition through corruption of underlying assembly instructions, that dont affect the loop counter. In this respect, Ravi *et al.* [RJH+19] proposed to compute the addition operation in the NTT domain (i.e.) compute $\mathbf{z}$ as $\mathsf{INTT}((\hat{\mathbf{s_1}} \circ \hat{c}) + \hat{\mathbf{y}})$. Thus, skipping fault in at least one coefficient of $\mathbf{z}$ uniformly propagates the fault to all coefficients through the subsequent INTT operation. This results in a invalid signature which is rejected by the conditional check on $\|\mathbf{z}\|_\infty$ with a very high probability (Line 27 in Alg.2). We refer to this combined countermeasure of using a dynamic loop counter along with addition in the NTT domain as the Verify_Add countermeasure.

3.Zeroization Faults: Bindel *et al.* [BBK16] proposed theoretical fault attacks to zeroize entire variables or a part of them to zero. They show that zeroizing the nonce $\mathbf{y}$ (Line 19) as well as the challenge polynomial c (Line 22) generates faulty signatures which easily compromise the secret key. We refer to these attacks together as the Zero_*Fault* attacks. Though theoretically possible, such zeroization of entire polynomials/modules is not trivial to achieve in practice, and the authors did not practically demonstrate such faults. We refer to these attacks together as the Zero_*Fault* attacks.

Countermeasure against Zeroization attack: Zeroization of $\mathbf{y}$ through skipping faults, can be protected using a well-designed loop counter protection, similar to that for the

Verify_Loop_Abort countermeasure. On the other hand, zeroization of the challenge polynomial c in the signing procedure leads to invalid signatures, which can be detected through the Verify_After_Sign countermeasure. Moreover, one can also explicitly check for such zeroization of variables through dedicated checking procedures.

3.1.2 Targeting the Verification Procedure

Unlike the signing procedure, the verification procedure has received much lesser attention with respect to fault injection attacks. However, bypassing the final verification operation (Line 39 in the Verify procedure of Alg.2) serves as a clear target for the attacker, and thus has to be protected. We are only aware of the work of Bindel *et al.* [BBK16], who showed that zeroization of the challenge polynomial c in the verification procedure, can lead to successful verification of invalid signatures for any message, without knowledge of the secret key. However, as stated earlier, such zeroization is not trivial to achieve in practice. Moreover, to the best of our knowledge, there has not been any practical fault injection attacks demonstrated on the verification procedure of Dilithium.

3.2 Fault Attacks on KEMs

We start by briefly describing application of Kyber KEM in a key-exchange protocol, before explaining the known fault attacks applicable to Kyber KEM. Refer Fig.2 for an example key-exchange protocol that can be built using IND-CCA secure Kyber KEM. The protocol is executed between two parties - Alice and Bob.

Alice starts by running the key-generation procedure (KeyGen) to generate her public-private key pair (pk, sk), and subsequently sends the public key pk to Bob. Bob then runs the encapsulation procedure (Encaps) procedure with the public key pk to generate the ciphertext ct and the session key K. Bob shares the ciphertext ct with Alice, who uses her secret key sk to generate the same shared session key K. Alice can choose to reuse the public-private key pair (pk, sk) for multiple key-exchanges and this is referred to as a static-key setting. However, Alice can also choose to use fresh key pairs (pk, sk) for every new key-exchange, which we refer to as the ephemeral-key setting. In this scenario, it is sufficient to perform key-exchange using the IND-CPA secure Kyber PKE. Here, Bob and Alice utilize the CPA.Encrypt and CPA.Decrypt procedures respectively, instead of the CCA.Encaps and CCA.Decaps to run the key-exchange protocol in the ephemeral-key setting.

If the fault attacker has physical access to Alice, he/she can target the key-generation and/or decapsulation procedure. If the attacker has physical access to Bob, then he/she can target the encapsulation procedure. In this work, we only consider attacks on the key-generation and encapsulation procedure, and thus attacks on the decapsulation procedure are considered out of scope.

3.2.1 Faulting the Key-Generation and Encryption/Encapsulation Procedure

The key-generation procedure is attractive for fault injection in an ephemeral setting, since it is performed for every new key exchange by Alice. Injection of faults in the key-generation procedure could lead to faulty public-keys that could easily compromise the secret key. An attacker can also target the encapsulation procedure through fault injection to produce faulty ciphertexts ct', which can compromise the corresponding secret message m. The encapsulation procedure is performed for every new key-exchange, and thus serves as an attractive target for the attacker for message recovery attacks.

In this respect, Ravi *et al.* [RRB+19] proposed the first practical fault attack for KEMs, targeting the key-generation and encryption procedure of schemes such as Kyber, NewHope and Frodo. The attack targets *byte sized nonces*, which are used during

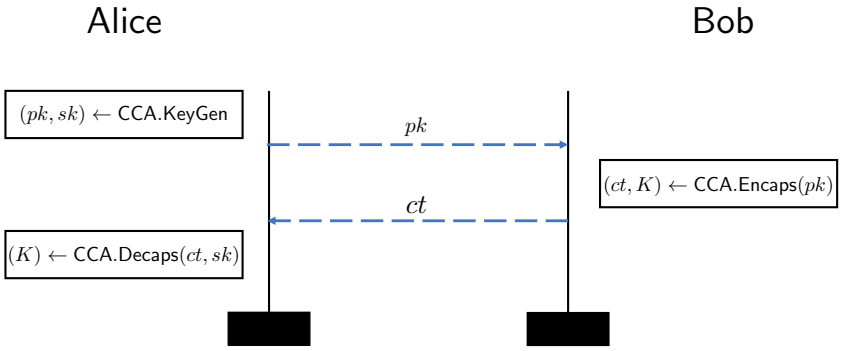


Figure 2: Key-Exchange protocol using IND-CCA secure Kyber KEM

sampling of secrets and errors to generate LWE instances. They demonstrate that a single or few targeted faults can be used to force nonce reuse, which results in sampling of the same/similar secrets and errors ($\mathbf{s} = \mathbf{e}$). This results in generation of weak LWE instances in the key generation and encryption procedure, that leads to trivial key recovery and message recovery attacks. We henceforth refer to this attack as the Nonce_Fault attack.

Countermeasure against Nonce_Fault *attack:* It is possible to utilize a dedicated verification procedure, which checks for the equality of the polynomials of the secret and error modules. Even if one of the polynomials of the secret/error is found to be equal, then the key-generation or encapsulation procedure can be aborted. Moreover, this dedicated verification procedure can be fortified with an additional loop counter protection so as to ensure that the verification procedure is not bypassed using a double fault attack (refer Section 3.1.1). A valid public key is generated at the output, only if the verification procedure is completed (successful loop counter check) and the verification procedure passes. On the other hand, a random public-key or ciphertext is generated as the output if the verification procedure is incomplete (failure in loop counter check) or the verification procedure fails. We refer to this countermeasure as the Verify_Nonce_Fault countermeasure.

3.2.2 Faulting the Decapsulation Procedure

To our knowledge, all remaining fault attacks applicable to lattice-based KEMs such as Kyber target the decapsulation procedure, to recover the long-term secret key.

Pessl and Prokop proposed a novel fault-assisted chosen-ciphertext attack [PP21] on Kyber KEM. Their attack works by injecting targeted faults in the message decoding operation, and subsequently utilizing information about the success/failure of decapsulation, as a decryption failure oracle. This information can be utilized by an attacker to recover the long term secret key in a few thousand chosen-ciphertext queries. While this attack can be thwarted by shuffling the message decoding operation, Hermelink *et al.* [HPP21] proposed an improved attack that can defeat the shuffling protection, but relies on a slightly stronger fault model of injecting targeted bit flip faults in memory. Delvaux [Del22] further improved the attack of Hermelink *et al.* [HPP21] by expanding the attack surface to several operations within the decapsulation procedure, while also working with a variety of more relaxed fault models. However, their attack requires a tens to thousands of fault to recover the secret key, as they rely on weaker and relaxed fault models.

Xagawa *et al.* [XIU+21] demonstrated that the obvious target of the final equality check in the decapsulation procedure can be easily skipped in several lattice-based KEMs such as Kyber KEM. The fault has the effect of downgrading the security of KEMs from CCA security to CPA security, which results in key recovery in a chosen-ciphertext setting. We do not delve deeper into attacks on the decapsulation procedure, as they are considered

out of scope of this work.

In summary, please refer to Tab.1 for a tabulation of the fault attacks and corresponding countermeasures for Kyber (key generation and encryption/encapsulation procedure) and Dilithium (signing procedure).

Table 1: Tabulation of known fault attacks and countermeasures for the key generation (KeyGen) and encapsulation (Encaps) procedure of Kyber KEM and signing procedure (Sign) of Dilithium.

Attack	Countermeasure
Kyber (KeyGen, Encaps)	
Nonce_Fault [RRB+19]	Verify_Nonce_Fault
Dilithium (Sign)	
Randomize_Secret_Key [BBK16, IMS+22]	Verify_After_Sign
Generic_DFA [BP18]	Verify_After_Sign
Loop_Abort [EFGT16]	Verify_Loop_Abort
Skip_Addition [BBK16, RJH+19]	Verify_Add
Zero_Fault [BBK16]	Verify_After_Sign,Verify_Loop_Abort

3.2.3 Motivation

Our survey of fault attacks on KEMs and signature schemes reveal that existing attacks are orthogonal in nature, with most attacks being specific either to KEMs or signature schemes. In this respect, we identify the Number Theoretic Transform (NTT) as a commonality, which is used in both lattice-based KEMs and signature schemes. It is a critical computational kernel, especially used in both Kyber and Dilithium to accelerate polynomial multiplication, and also manipulates sensitive intermediate variables including secret keys. The NTT has been target of several side-channel attacks [PPM17a, PP19, MBM+22, HHP+21]. The inherent properties of the NTT have also been exploited to perform cold-boot attacks [ADP18], where the attacker attempts to recover the secret key of LWE-based schemes from noisy versions of the key. However, there are no known studies on understanding the vulnerability of NTT to fault injection attacks. Moreover, a fault vulnerability exploiting the inherent nature of the NTT, if exists, can be potentially used to exploit multiple post-quantum cryptographic schemes, which utilize NTT for fast and efficient polynomial multiplication.

Given its widespread usage in several schemes, it becomes imperative to analyze its susceptibility to FIA and identify suitable countermeasures for protection. Moreover, the uniform structure of the NTT based on the "butterfly" network, makes it especially interesting to analyze from the perspective of fault injection attacks. Thus in this work, we perform the first fault injection analysis of the NTT, and analyze its applicability to lattice-based schemes. We identify a critical vulnerability within the NTT operation, which can be exploited in a variety of different open-source software implementations from independent designers. Moreover, all our proposed attacks only require a single targeted fault to be injected within the target procedure. Moreover, we also show that our attacks can bypass most fault countermeasures against existing attacks on lattice-based KEMs and signature schemes.

4 Fault Vulnerability of NTT

4.1 Intuition

We start by analyzing a single CT butterfly operation (described in Eqn.4), commonly used to implement the forward NTT. Its inputs are $(x_0, x_1) \in \mathbb{Z}_q$, twiddle constant w, and outputs are $(y_0, y_1) = ((x_0 + x_1 \cdot w), (x_0 - x_1 \cdot w))$. We consider the possibility of injecting faults to zeroize the twiddle factor w. As a result, the faulty outputs of the butterfly are $(y_0^*, y_1^*) = (x_0, x_0)$, with no effect of x_1 on the faulty output. We now extend the same fault to all the butterflies in a single stage of NTT (Refer to Stage-1 of NTT in Fig.1). Let the input to the stage be x_i for $i \in [0, n-1]$ and its output be y_i for $i \in [0, n-1]$. If all of its twiddle constants are 0, then the output is given as:

$$y_i = \begin{cases} x_i, & \text{for } i < n/2 \\ x_{i-(n/2)}, & \text{otherwise} \end{cases} \tag{6}$$

We observe that the entropy of the output is reduced by half. If we extend the same fault to the entire NTT, then the final output of NTT $\hat{x}$ is simply $\hat{x}_i = x_0 \, \forall \, i \in [0, n-1]$. In essence, the entropy of the output is reduced by half for every stage, with the final output only containing a single element x_0 repeated n times. Thus, zeroizing the twiddle constants produces a faulty output with very low entropy. If this faulty NTT output is utilized for polynomial multiplication with $z \in R_q$ (i.e.) $x \cdot z \in R_q$, then the faulty product is $x^* \cdot z \in R_q$ where x^* is given as,

$$x_i^* = \begin{cases} x_0, & \text{if } i = 0 \\ 0, & \text{otherwise} \end{cases} \tag{7}$$

Thus, faulting the NTT of x in this manner has the effect of *implicitly changing* x to x^* with low entropy, with only a single non-zero coefficient. While this applies for schemes such as Dilithium which utilize a complete NTT, Kyber utilizes an incomplete NTT with last stage skipped. The implicitly modified faulty input x^* in case of Kyber KEM is given as:

$$x_i^* = \begin{cases} x_i, & \text{for } i = \{0, 1\} \\ 0, & \text{otherwise} \end{cases} \tag{8}$$

with two non-zero coefficients. Thus, the entropy of the faulty input x^* depends upon the number of stages in the NTT. *While zeroization of all the twiddle constants comes across as a strong assumption, we have identified a critical fault vulnerability in practical implementations of NTT in several schemes, which enables zeroization of twiddle constants with only a single targeted fault.*

4.2 Analyzing Practical NTT Implementations

We utilize the optimized implementation of Kyber KEM from the *pqm4* library for 32-bit ARM Cortex-M4 based microcontrollers [KRSS19] for our analysis[2]. We compiled our implementations using the `arm-none-eabi-gcc` compiler, with the highest compiler optimization level `-O3`. We analyzed the compiled assembly code using an On-Chip Debugger to better understand the utilization of twiddle constants within the NTT/INTT computation.

[2]Our analysis and experiments were carried out on the NTT implementations of Kyber and Dilithium corresponding to the commit hash `cf6f358c05db8a4e416561801bb4920d05b3bbb1`, and were available in the *pqm4* library until Jan 31, 2022. However, our attacks also apply in the same manner to the most recent NTT implementations in the *pqm4* library.

Algorithm 3 Assembly Optimized NTT of Kyber in *pqm4* library [KRSS19] (Simplified)

```
1:  ldr r1, [pc, #4]                    ▷ Loading twiddle-ptr from address (pc+4) to register r1
2:
3:                                      ***Start of NTT Assembly Routine***
4:
5:  n ← 16
6:  while n > 0 do                      ▷ First stage (Stage 1,2,3)
7:      load poly
8:      ldrh twiddle, [twiddle-ptr]     ▷ Loading twiddle from twiddle-ptr
9:      doublebutterfly (poly, twiddle)
10:     ldr twiddle, [twiddle-ptr, #2]  ▷ Loading twiddle from (twiddle-ptr+2)
11:     doublebutterfly (poly, twiddle)
12:     ···
13:     n − −
14: end while
15: add twiddle-ptr, #14                ▷ Incrementing twiddle-ptr by 14 for next stage
16: n ← 8
17: while n > 0 do                      ▷ Second stage (Stage 4,5,6)
18:     m ← 2
19:     while m > 0 do
20:         load poly
21:         ldrh twiddle, [twiddle-ptr] ▷ Loading twiddle from twiddle-ptr
22:         doublebutterfly (poly, twiddle)
23:         ldr twiddle, [twiddle-ptr, #2]  ▷ Loading twiddle from (twiddle-ptr+2)
24:         doublebutterfly (poly, twiddle)
25:         ···
26:         m − −
27:     end while
28:     add twiddle-ptr, #14            ▷ Incrementing twiddle-ptr by 14 for next stage
29:     n − −
30:     ···                             ▷ Last stage (Stage 7)
31: end while
```

Refer Alg.3 for a simplified pseudo-code of the assembly optimized NTT routine of Kyber. The twiddle constants are pre-computed and stored as a *constant* array at a particular address in the flash memory (during compile time), denoted as T. This base address T of the twiddle constant array is also stored as a 32-bit value at a given location in the flash memory. Once the NTT routine is called, the base address T is first loaded from flash memory (in our case, the address is $(\mathsf{pc}+4)$ where pc is the program counter) into register $r1$ using the ldr instruction (Line 1 colored in red). The base address T in r1 is then used as a pointer to reference different constants in the twiddle constant array (Lines 8,10,15,21,23,28 colored in orange). We therefore refer to T as the twiddle pointer.

We make a key observation that the address for all the twiddle constants are calculated using the twiddle pointer T. If an attacker can fault the twiddle pointer from T to T^* (Line 1), then all the twiddle constants for the NTT are retrieved from a modified address T^*. If T^* points to a memory location filled with zeros, then all the twiddle constants are essentially zeroized with only a single fault. This, therefore serves as a *single point of failure* to zeroize all the twiddle constants of a target NTT, which we refer to as the twiddle-pointer vulnerability of the NTT.

To zeroize the twiddle constants using a single fault, there are two main conditions:

1. Condition-1: Fault the twiddle pointer T to T^*, when loaded from flash memory.
2. Condition-2: The faulty twiddle pointer T^* points to an array filled with zeros.

4.2.1 Condition-1: Faulting Data Loaded from Flash Memory

Faulting the data loaded from flash memory to the register was first reported by Menu et al. [MBD+19] using Electromagnetic Fault Injection (EMFI) on an ARM Cortex-M3 based microcontroller. They demonstrated the ability to perform both bit-set and bit-reset faults on the fetched data, at a byte-level precision with up to 100% repeatability. They show that the data prefetch buffer which is loaded with data from flash memory is sensitive to fault injection. Thus, the faults are not injected on static data stored in flash memory, but data in transit, when loaded from the flash memory to the registers.

The same fault model has also been used in a recent work by Soleimany *et al.* [SBH+22] on the ARM Cortex-M4 microcontroller, to demonstrate Persistent Fault Analysis on block ciphers using EMFI. As we show later in Sec.7, we were also able to achieve the same fault model on a similar ARM Cortex-M4 device with a very high repeatability (up to 100%).

4.2.2 Condition-2: Retrieving Zero Data from Memory Access

We also require that the memory accesses from the faulty twiddle pointer T^* results in fetch of a zero twiddle constant array. This naturally raises a question of *how many locations in the target's addressable memory result in fetch of a zero array.* We therefore performed an empirical memory analysis on our DUT, (i.e.) STM32F407VG microcontroller (ARM Cortex-M4), to estimate the probability of fetching a zero array from a random 32-bit address. For each memory access, there are three possible outcomes: (1) Zero array - Success (2) Non-zero array - Failure and (3) Hard Fault due to illegal memory access - Failure. In several instances, we also observe that the CPU can fetch zero data, even if the faulty address is not mapped to a physical memory such as Flash/SRAM. For $10k$ random memory accesses, we obtained a reasonably high success rate of $\approx 20-25\%$ to retrieve a zero twiddle constant array. Only $\approx 0.1\%$ memory accesses led to retrieval of non-zero data, while all the remaining memory accesses led to a hard fault, where the device becomes unresponsive. We do agree that the obtained numbers are not fixed for a given device, but significantly depend upon factors such as memory initialization, memory utilization by concurrent software or other peripherals etc. We performed our experiments on our DUT (i.e.) STM32F407 MCU, which only runs the memory test program and the DUT has been configured based on the *pqm4* library. But, what is interesting to note is that retrieval of data from memory locations that are not mapped to any physical memory location also return zeros.

After identifying fault parameters that satisfy both the conditions with high repeatability, during an initial profiling, the attacker can achieve 100% attack success as shown later in Sec. 7. Our practical experiments yield a very high fault repeatability (up to 100%) to zeroize all the twiddle constants using a single fault in both Kyber and Dilithium.

4.2.3 On targeting the input to the NTT:

Our analysis of the NTT implementations in Kyber and Dilithium revealed that coefficients of the NTT input are also accessed using a single pointer variable (denoted as input pointer P). On first glance, it might appear that a single fault on the input pointer P can also zeroize the entire NTT input. However, this pointer is not susceptible to EMFI, at least in the same manner as the twiddle pointer. Unlike the *constant* twiddle array, whose pointer/address is fetched from the flash memory, the NTT input is a *variable* whose address P is calculated on the fly using arithmetic instructions, and not fetched from flash memory. Thus, the input pointer P is not exposed to EMFI in the same way as the twiddle pointer T. Moreover, it is not clear how P can be faulted using EMFI or other attack vectors.

Even if the attacker can fault the input pointer, there are significant challenges. The input pointer P is dynamically computed several times within a single execution of the

target procedure (key generation, encapsulation, signing or verification) in Kyber/Dilithium. So, all the computations pertaining to P need to be faulted to ensure that the faulty value is used throughout the computation. Otherwise, the faulty output is of no use to the attacker. Establishing such precise synchronization to fault every time instance when P is manipulated is very hard to achieve in practice. This also requires a very detailed knowledge about the implementation at the assembly level. On the other hand, faulting the pointer to twiddle factors only needs to be done once for the target NTT operation. Thus, we argue that NTT's twiddle pointer serves as an easier target for an attacker for practical fault injection attacks.

4.2.4 On Pointer Protection/Integrity in Software Implementations

There are several works that have demonstrated the ability to exploit pointer manipulation to perform control flow hijacking attacks [Sha07, CXS+05, HSA+16]. These are typically software attacks which typically exploit memory corruption vulnerabilities such as buffer overflows. Thus, there are several proposals such as software stack protection, randomized stack/heap, address space layout randomization (ASLR) and pointer authentication [LNW+19, YPK+22, SMD+13]. While these mitigation techniques typically cater to software attacks, it is not clear if they can also prevent hardware attacks such as transient fault injection. Moreover, these protections are typically available in the higher-end processors such as x86-64 and ARM Cortex-A based processors, while we are not aware of such countermeasures in embedded microcontrollers such as ARM Cortex-M based devices.

A designer can implement dedicated and custom software countermeasures to protect data pointers, particularly for cryptographic implementations. In this respect, utilization of parity checks or dummy registers for redundant loading of target data pointer could be used as potential countermeasures. However, it is possible that the designer only chooses to protect pointers to sensitive data such as secret keys and sensitive intermediate variables, while not choosing to protect pointers to public data such as public keys and constants. We however demonstrate that manipulating public data such as twiddle factors can also lead to devastating attacks compromising the security of lattice-based KEMs and signature schemes. Moreover, we are not aware of any prior work that faults data pointers to a memory location filled with zeros, at least in the context of attacks on post-quantum cryptographic schemes.

4.2.5 Analysis of Multiple Open-Source Implementations

We performed an analysis of several independently designed optimized implementations of the NTT for Kyber and Dilithium, for our DUT (i.e.) ARM Cortex-M4 based microcontroller. Our motivation was to analyze the manipulation of $\mathsf{twiddle-pointer}$ in different NTT implementations. We can positively confirm that the same $\mathsf{twiddle\text{-}pointer}$ vulnerability could be identified in three different NTT implementations of Kyber based on the works of Botros *et al.* [BKS19], Alkim *et al.* [ABCG20] and Amin *et al.* [AHKS22]. We also observed the same behaviour in the NTT implementations for Dilithium, based on the works of Guneysu *et al.* [GKOS18], Greconici *et al.* [GKS21] and Amin *et al.* [AHKS22]. Thus, the $\mathsf{twiddle\text{-}pointer}$ has been manipulated in the same manner, across several independently developed optimized implementations of NTT, which result in the presence in the $\mathsf{twiddle\text{-}pointer}$ vulnerability in all the assembly optimized implementations of NTT for the ARM Cortex-M4 microcontroller.

4.3 Applicability to SCA Countermeasures

We now discuss the applicability of our attack to SCA countermeasures such as shuffling and masking.

4.3.1 Applicability to Shuffling Countermeasures

Shuffling countermeasures are commonly used in varying degrees to protect against side-channel attacks and fault injection attacks. Our proposed attack only requires to corrupt the twiddle factor data through pointer manipulation. Thus, shuffling the order of operations before, after or during the target NTT operation does not deter our attack, as long as the twiddle factors are zeroized. Recently, Ravi *et al.* [RPBC20] proposed shuffled NTT implementations, intended to protect against single trace side-channel attacks [PP19, PPM17b]. Validating our hypothesis, we experimentally verified that the shuffled NTT implementation can be attacked in the same manner as the unprotected NTT,as shown later in Section 7.4.1.

4.3.2 Applicability to Masking Countermeasures

Masking countermeasures for Kyber and Dilithium against SCA typically work by additively splitting the secret into multiple shares and independently computing over them. We consider a typical masked implementation of Kyber with t shares (i.e.) protected against SCA attack of order $(t-1)$. The secret $x \in R_q$ is additively split in the following manner: $x = (x1 + x2 + \ldots + xt)$ where $xi \in R_q$ for $i \in [1, t]$ denote the t additive shares. Thus, NTT over x is computed by performing NTT over all the individual shares (i.e.) $\mathsf{NTT}(xi)$ for $i \in [1, t]$ since $\mathsf{NTT}(x) = \sum_{i=(t)}^{i=1} \mathsf{NTT}(xt)$. Let us consider the effect of faulting the twiddle factors of all these NTTs to zero. Thus, the coefficients of the faulty NTT output of each share xi for $i \in [1, t]$ is given as:

$$(\hat{xi})_j = (xi)_0, \ \forall \ j \in [0, n-1] \tag{9}$$

where $(xi)_0$ denotes the first coefficient of the polynomial xi. Thus, the coefficients of the faulty NTT output of the unshared secret x is nothing but:

$$(\hat{x})_j = (x1)_0 + (x2)_0 + \ldots + (xt)_0, \ \forall \ j \in [0, n-1] \tag{10}$$

If the faulty NTT shares are used for polynomial multiplication, then the modified faulty secret x^* is nothing but,

$$x_i^* = \begin{cases} (x1)_0 + (x2)_0 + \ldots + (xt)_0, & \text{if } i = 0 \\ 0, & \text{otherwise} \end{cases} \tag{11}$$

where the sum $((x1)_0 + (x2)_0 + \ldots + (xt)_0)$ is nothing but x_0. Thus, faulting the NTT of all the individual shares of x generates the same output as faulting the NTT over the unshared polynomial x (Refer Eqn.7). We have experimentally validated the above concept through fault simulations for the NTT used in both Kyber and Dilithium. Thus, our proposed attack is also applicable to masked implementations, albeit with increase in the number of targeted faults depending upon the number of shares t of the target polynomial.

5 Practical Attacks on Kyber

In this section, we propose novel key recovery and message recovery attacks on Kyber exploiting the twiddle-pointer fault vulnerability. Our analysis utilizes the algorithm of CPA secure PKE of Kyber in Alg.1 for explanation. We utilize the key exchange protocol

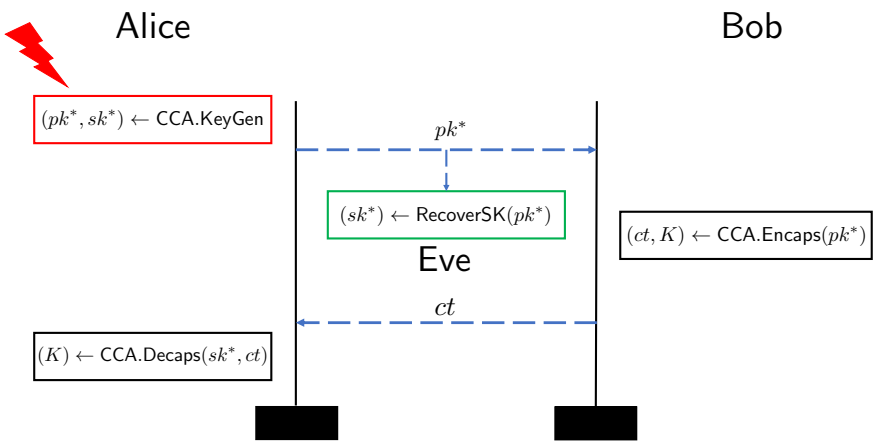


Figure 3: Kyber-Key-Recovery attack on key exchange protocol built upon IND-CCA secure Kyber KEM

described in Fig.2, run between Alice and Bob to describe our key recovery and message recovery attacks. Eve is the attacker who targets the key-generation procedure of Alice for key recovery or the encryption/encapsulation procedure of Bob for message recovery.

5.1 Targeting Key Generation for Key Recovery

Our key recovery attack targets the NTTs in the key generation procedure, to generate public keys whose secret keys have a very low entropy. We propose to fault the NTT operation on the secret $\mathbf{s} \in R_q^k$ (Line 7). Let the faulty NTT output be denoted as $\hat{\mathbf{s}}^*$. Since $\hat{\mathbf{s}}^*$ is utilized to generate the LWE instance (Line 9), the LWE instance is implicitly created using a low-entropy secret $\mathbf{s}^*$. If all the k NTTs of $\mathbf{s}$ are faulted, then only the first two coefficients of every polynomial of $\mathbf{s}^*$ are non-zero, while all the other coefficients are zeros. For Kyber768 with $k = 3$ and the span of the coefficients in $[-2, 2]$, the faulty secret key $\mathbf{s}^*$ can be recovered from the public key with a brute-force complexity of 5^6 ($= 15{,}625$). We can utilize the following approach to arrive at the exact value of $\mathbf{s}^*$. For each guess of $\mathbf{s}^*$, we can compute the difference $\mathbf{d} = \mathbf{t} - \mathbf{A} \cdot \mathbf{s}^*$. The difference $\mathbf{d}$ for the correct guess will have a short span equal to that of the error of the LWE instance (i.e.) $[-2, 2]$. Once the target NTTs are faulted, the secret key can be recovered with a 100% success rate. We henceforth refer to this as the Kyber-Key-Recovery attack. Refer to Fig.3 for an illustration of the Kyber-Key-Recovery attack on the key exchange protocol built upon IND-CCA secure Kyber KEM. We denote the attacker as Eve who faults the key-generation procedure of Alice (highlighted in red) for key recovery.

Since the secret key of Kyber is stored in the NTT domain, the same faulty secret is also used in the decryption procedure. Thus, the injected fault in the key generation procedure also propagates to the decryption procedure. Moreover, the faulty secret $\mathbf{s}^*$ is also valid, since $\ell_\infty(\mathbf{s}^*)$ respects that of a valid secret of Kyber. Thus, key recovery is successful while also maintaining the correctness of the scheme. Since the faulty public key is a valid LWE instance, it is indistinguishable from random, making it difficult to detect our attack, simply from analyzing the public key.

5.2 Targeting Encryption for Message Recovery

Our message recovery attack targets the encryption procedure of Kyber KEM. The aim is to recover the message from a valid ciphertext corresponding to a key exchange between two parties (Alice and Bob). We propose to target the NTT of $\mathbf{r}$ in the encryption procedure (Line 17), which ensures use of a low-entropy $\mathbf{r}^*$ to generate the ciphertext. Similar to

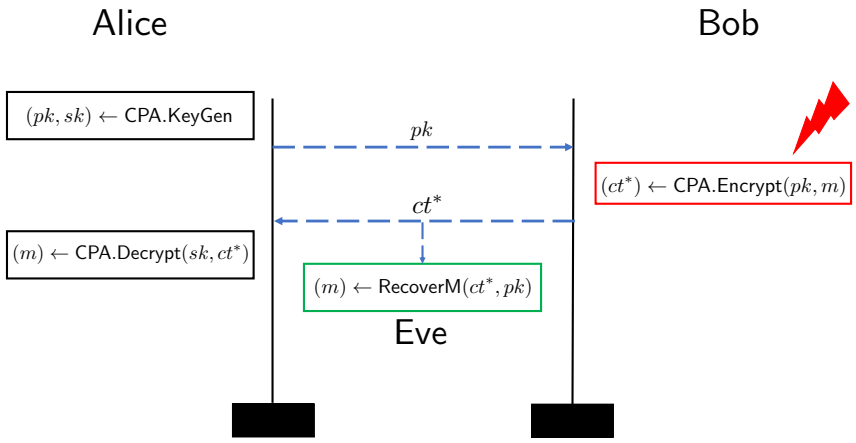


Figure 4: Kyber-Message-Recovery attack on key exchange protocol built upon IND-CPA secure Kyber PKE

our key recovery attack, the brute-force complexity to guess $\mathbf{r}^*$ for Kyber768 is 5^6. If the correct $\mathbf{r}$ can be recovered, the secret message m can be recovered from the faulty ciphertext $(ct^* = (\mathbf{u}^*, v^*))$ as follows:

$$m = \mathsf{Compress}(v^* - \mathsf{INTT}(\hat{\mathbf{t}} \circ \mathsf{NTT}(\mathbf{r})), 1)$$

Among the 5^6 possibilities for $\mathbf{r}^*$, the correct value of $\mathbf{r}^*$ can be recovered as follows. For a given guess of $\mathbf{r}^*$, the erroneous message polynomial can be calculated as,

$$\mathbf{m} = \mathbf{v}^* - \mathsf{INTT}(\mathbf{t}^T \circ \mathbf{r}^*)$$

For the correct guess, the coefficients of $\mathbf{m}$ are clustered around 0 and $q/2$ with a short span, while for all other guesses, the coefficients are uniformly distributed in $\mathbb{Z}_q$. Once the target NTTs are faulted, the message can be recovered with a 100% success rate. We henceforth refer to this as the Kyber-Message-Recovery attack. The impact of our message recovery attack depends upon whether the attacker targets the (1) CPA Secure PKE or (2) CCA secure KEM of Kyber.

5.2.1 Attacking CPA secure Kyber PKE

The CPA secure PKE is typically used for ephemeral key exchanges. Faulting its encryption procedure results in creation of a faulty ciphertext. However, the faulty ephemeral secret $\mathbf{r}^*$ used to generate the ciphertext is valid, since $\ell_\infty(\mathbf{r}^*)$ respects that of a valid ephemeral secret. Since the decryption procedure does not check for the validity of the ciphertext, the correctness of key exchange is maintained, while also resulting in message recovery. Refer to Fig.3 for an illustration of the Kyber-Message-Recovery attack on the key exchange protocol based on IND-CPA secure Kyber PKE. The attacker Eve faults the encryption procedure of Bob (highlighted in red) for message recovery.

5.2.2 Attacking CCA secure Kyber KEM

The decapsulation procedure of CCA secure Kyber can detect the validity of a ciphertext with a very high probability. Thus, the faulty ciphertext is rejected by the decapsulation procedure. This is because the ephemeral secret $\mathbf{r}$ used in the encryption procedure (Alice) differs from that used in the re-encryption procedure after decryption (Bob). This leads to

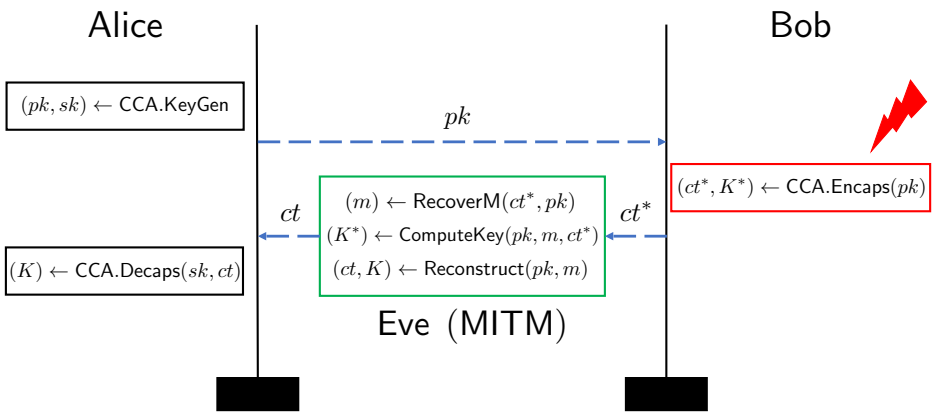


Figure 5: Kyber-Message-Recovery attack on key exchange protocol built upon IND-CCA secure Kyber KEM

failure of the key exchange, thereby rendering message recovery useless. The attack only works if the attacker can fault the NTT of $\mathbf{r}$ in both the encapsulation and decapsulation procedure. However, this is a very strong assumption since the attacker requires access to both the communicating devices for a successful attack.

But, we observe that a Man-In-The-Middle (MITM) attacker can perform message recovery, while still ensuring the correctness of key exchange between Alice and Bob. Refer to Fig.4 for the Kyber-Message-Recovery attack on the key exchange protocol based on IND-CCA secure Kyber KEM. Eve faults Bob's encapsulation procedure (highlighted in red), and also serves as the MITM between Alice and Bob.

The attack is carried out as follows: Faulting Bob's encapsulation procedure results in generation of a faulty and invalid ciphertext ct^* for message m and the corresponding session key of Bob is K^*. Eve (MITM) uses the faulty ciphertext ct^* and the public key pk to recover the message m (denoted as RecoverM). The knowledge of m, pk and ct can be used to compute the session key of Bob by simply running the appropriate operations of the decapsulation procedure (i.e.) K^* (denoted as ComputeKey). Subsequently, the knowledge of pk and m can be used to generate a valid ciphertext ct for m by running the encapsulation procedure and the corresponding session key K (denoted as Reconstruct). Eve then sends the valid ciphertext ct to Alice, who decapsulates using the secret key to generate the same session key K. Thus, Eve has the knowledge of session keys of both Alice (K) and Bob (K^*), and therefore can decrypt all communication between Alice and Bob.

6 Practical Attacks on Dilithium

In this section, we demonstrate two types of attacks on Dilithium exploiting the twiddle-pointer vulnerability: (1) Existential Forgery Attack and (2) Verification Bypass Attack. We utilize the algorithm of Dilithium in Alg.2 for our analysis.

6.1 Existential Forgery Attack

An attacker can forge signatures of Dilithium, if he/she is able to retrieve its primary secret $\mathbf{s_1}$. A close observation of the signing procedure reveals that the primary signature component $\mathbf{z}$ is closely dependent on s_1, and thus faulting the generation of $\mathbf{z}$ can reveal

information about $\mathbf{s_1}$. Generation of $\mathbf{z}$ (Line 24) is done as follows:

$$\begin{aligned}\mathbf{z} &= \mathsf{INTT}(\mathsf{NTT}(\mathbf{s_1}) \circ \mathsf{NTT}(c)) + \mathbf{y} \\ &= \mathsf{INTT}(\hat{\mathbf{s_1}} \circ \hat{c}) + \mathbf{y}\end{aligned} \tag{12}$$

Essentially, $\mathbf{z}$ is nothing but the ephemeral nonce $\mathbf{y}$, additively masked by the product $\mathbf{s_1} \cdot c$, where c is public and is part of the signature. For brevity, we refer to $\mathbf{s_1}$ as $\mathbf{s}$. For simplicity, our analysis assumes all operands in Eqn.12 are single polynomials in R_q. Since the polynomials in each operand are handled independently of each other, our analysis can be easily extended to all the polynomials in a straightforward manner. We present two novel key recovery attacks on both the deterministic and probabilistic/randomized variants of Dilithium. We assume that the attacker can trigger the target device to generate signatures for any message of his/her choice.

6.1.1 Attack-1: Targeting Deterministic Dilithium

Our first attack is a differential style fault attack targeting the signing procedure of deterministic Dilithium. Our target is the NTT of the challenge polynomial c. We recall that the challenge polynomial c is sparse with coefficients in $\{-1, 0, 1\}$, and the coefficients of c are represented as $(c_0, c_1, c_2, \ldots, c_{n-1})$. Our attack is carried out as follows: The attacker lets the target sign the message m, whose correct signature is $\sigma = (\mathbf{z}, \mathbf{h}, c)$. The message m is chosen such that the first coefficient of challenge c is 0 (i.e.) $c_0 = 0$. The attacker yet again lets the target sign m, but this time, the NTT of c is faulted to zeroize all its twiddle constants. As a result, the faulty $c^* = (c_0, 0, 0, \ldots, 0)$. Since $c_0 = 0$, the faulty challenge $c^* = 0$. As a result, the faulty signature $\mathbf{z}^*$ is given as:

$$\begin{aligned}\mathbf{z}^* &= \mathbf{s} \cdot c^* + \mathbf{y} \\ &= \mathbf{y} \quad (\because c^* = 0)\end{aligned} \tag{13}$$

which is nothing but the ephemeral nonce $\mathbf{y} \in R_q^\ell$. Thus, the difference between $\mathbf{z}$ and $\mathbf{z}^*$ ($\Delta\mathbf{z}$) simply yields the product $\mathbf{s} \cdot c$. Since c is known, $\mathbf{s}$ can be easily calculated as $\Delta\mathbf{z} \cdot c^{-1} \in R_q$.

The signing procedure follows the Fiat-Shamir with Aborts framework and thus presents additional challenges. Successful key recovery requires that both the valid and faulty signatures utilize the same number of iterations (κ) before exiting the abortion loop (i.e.) $\Delta(\kappa) = \kappa^* - \kappa = 0$. However, the use of faulty intermediate values do not always guarantee termination at the same iteration. Thus, not all successful faults result in key recovery.

We therefore performed empirical fault simulations using 1000 secret keys, assuming a perfect fault on the NTT of c. We observed that an average of ≈ 13 signatures are enough to recover the secret key with 100% success rate. We henceforth refer to this as the Sign_Fault_NTT_C attack. Since the generated faulty signatures are invalid, verification after sign serves as an effective countermeasure against this attack. The same attack does not work on the probabilistic signing procedure since it is a differential style fault attack.

6.1.2 Attack-2: Targeting Probabilistic Dilithium

The probabilistic signing procedure of Dilithium samples a random ephemeral nonce $\mathbf{y}$ for every execution (independent of the message m). This makes it impossible to know *apriori*, the number of iterations of the signing procedure for a given message m. Combined with the influence of non-constant time rejection checks, the operations in the signing procedure are *temporally randomized*, which makes it very difficult to perform injected targeted faults. Moreover, differential style fault attacks do not apply, since the computations are also randomized. Thus, mounting practical fault injection attacks on probabilistic Dilithium is very challenging, especially using targeted faults. We however show that the

twiddle-pointer vulnerability can be exploited for key recovery over probabilistic Dilithium in certain settings.

The main target of our attack is the NTT over the ephemeral nonce $\mathbf{y}$ (Line 20). However, we observe that the current implementations of Dilithium calculate the primary signature $\mathbf{z}$ using $\mathbf{y}$ in the normal domain (Line 24). Thus, faulting the NTT of $\mathbf{y}$ does not reveal any information about $\mathbf{s_1}$. However, computing $\mathbf{z}$ in this manner is merely an implementation choice and that $\mathbf{z}$ can be alternatively computed as

$$\begin{aligned} \mathbf{z} &= \mathsf{INTT}(\mathsf{NTT}(\mathbf{s_1}) \circ \mathsf{NTT}(c) + \mathsf{NTT}(\mathbf{y})) \\ &= \mathsf{INTT}(\hat{\mathbf{s_1}} \circ \hat{c} + \hat{\mathbf{y}}) \end{aligned} \tag{14}$$

Generating $\mathbf{z}$ in this manner also serves as a countermeasure against Skip_Add fault injection attacks (Refer Section 3). Moreover, it also has an advantage of not requiring to retain/store $\mathbf{y}$ in memory, thereby reducing dynamic memory consumption by about 3.68 KB for Dilithium3. We are not aware of a public implementation of Dilithium adopting this approach. Nevertheless, this alternative approach is indeed attractive for an designer as a memory optimization technique as well as protection against Skip_Add fault injection attacks. We however identify that utilizing $\mathsf{NTT}(\mathbf{y})$ to generate $\mathbf{z}$ in the aforementioned manner, makes it possible to also target probabilistic Dilithium for key recovery in the following manner.

Firstly, operations in the probabilistic signing procedure are temporally randomized. We however observe that the NTT of $\mathbf{y}$ (Line 20) is performed before the first rejection check (Line 27). Thus, the NTT of $\mathbf{y}$ in the first iteration, always occurs at a fixed time, from the start of the signing procedure, thereby making it possible to be targeted through fault injection. By faulting the NTT of $\mathbf{y}$, $\mathbf{z}$ is computed using a low-entropy $\mathbf{y}^*$ and the faulty signature $\mathbf{z}^*$ is given as:

$$\mathbf{z}^*[i] = \begin{cases} \mathbf{sc}[i] + \mathbf{y}[i], & \text{for } i = 0 \\ \mathbf{sc}[i], & \text{for } 1 \leq i < n-1 \end{cases} \tag{15}$$

where $\mathbf{sc}$ is the product $\mathbf{s} \cdot c$. Thus, all but the first coefficient of $\mathbf{sc}$ are exposed as part of the faulty signature $\mathbf{z}^*$. An attacker can simply guess the first coefficient of $\mathbf{sc}$ and subsequently calculate $\mathbf{s}$ for each guess, until he/she finds out the correct $\mathbf{s}$. The correct $\mathbf{s}$ can be found out by simply checking if the span of the recovered $\mathbf{s}$ (i.e.) $\ell_\infty(\mathbf{s})$ satisfies the bounds of a valid secret. A wrong guess will simply yield an $\mathbf{s}$ with a very large ℓ_∞ norm. For successful key recovery, the faulty signature and its associated intermediate variables should also satisfy all the rejection checks of the abortion loop.

We performed empirical fault simulations using 1000 secret keys and an average of ≈ 3 signatures are sufficient to recover the secret key with 100% success rate. To the best of our knowledge, we have presented the first practical fault injection attack applicable to the probabilistic variant of Dilithium, resulting in full key recovery without requiring any brute-force search. We henceforth refer to this attack as Sign_Fault_NTT_Y. This attack also works on the deterministic variant of Dilithium. Moreover, the faulty signature generated using the low entropy nonce $\mathbf{y}^*$ is valid and thus passes verification. Thus, the verification after sign countermeasure does not work against this attack, which makes it a more stealthier attack compared to the Sign_Fault_NTT_C attack.

6.2 Verification Bypass Attack

While the aforementioned attacks target the signing procedure, the verification procedure also serves as a good target for fault injection attacks. One of the main motivation being, forceful acceptance of invalid signatures through faults, for any message of the attacker's choice. One of the obvious and known targets for fault injection is to simply skip the

final comparison operation that decides the validity of the received signatures (Line 39). So, it is possible that the designer fortifies the comparison operation to protect against such trivial attacks. Bindel *et al.* [BBK16] proposed a novel *zeroing fault* attack on the verification procedure of GLP and BLISS signature schemes. They show that zeroizing the challenge c during verification can force acceptance of invalid signatures. However, faulting an entire polynomial to zero is very difficult to achieve in practice. Moreover, the applicability of their attack to Dilithium is also not clear, considering the underlying differences between the signature schemes. In the following, we demonstrate exploitation of the twiddle-pointer fault vulnerability to present the first practical *zeroing fault* attack on the verification procedure of Dilithium.

For a given signature $\sigma = (\mathbf{z}, \mathbf{h}, c)$, the verification procedure computes $\mathbf{w_1}'$ (Line 37), which is further hashed with the message μ to recompute the challenge $\bar{c}$ (Line 38). Then, $\bar{c}$ is compared with the received challenge polynomial c, and the result of comparison determines validity of the signature. The main target of our attack is the NTT operation over c (Line 36). If $c_0 = 0$, then faulting the NTT of c ensures that a faulty $\hat{c}^* = 0$ is used to compute a faulty $\mathbf{w_1}^*$, which is given as:

$$\begin{aligned} \mathbf{w_1}^* &= \mathsf{UseHint}(\mathbf{h}, \mathbf{A} \cdot \mathbf{z}) \\ c^* &= \mathcal{H}(\mu \| \mathbf{w_1}^*) \end{aligned} \tag{16}$$

We observe that faulty $\mathbf{w_1}^*$ is only dependent on $(\mathbf{h}, \mathbf{z})$, which an attacker is free to choose. We therefore propose to generate a malicious signature in the following manner: the attacker samples a random $(\mathbf{z}^*, \mathbf{h}^*)$ whose respective norms satisfy the conditions for successful verification. For a chosen message μ, he/she computes $\mathbf{w_1}^*$ and c^* as in Eqn.16, and repeats as until $c_0^* = 0$. Then, the attacker's crafted signature for μ is $\sigma^* = (\mathbf{z}^*, \mathbf{h}^*, c^*)$. Refer Alg.4 for an algorithmic description to create a malicious signature for our verification bypass attack.

In the attack phase, the attacker queries the verification procedure with (σ^*, μ) and faults the NTT over c^*. Since $c_0^* = 0$, the injected fault zeroizes the challenge c and thus computes the same $\mathbf{w_1}^*$ and challenge c^*, thereby resulting in successful verification. We performed empirical fault simulations using 1000 random messages and were able to enforce acceptance of invalid signatures for all the messages, thereby demonstrating a 100% success rate for our verification bypass attack. We henceforth refer to this as Verification-Bypass attack.

Algorithm 4 Malicious Signing Procedure for Verification Bypass Attack

1: **procedure** MALICIOUS-SIGN(sk, M)
2: $\quad\hat{\mathbf{A}} \in R_q^{k \times \ell} \leftarrow \mathsf{Expand}(seed_A)$
3: $\quad\mu \in \{0,1\}^{512} \leftarrow \mathcal{H}(tr \| M)$
4: $\quad$**while** $c_0 = 0$ **do** ▷ Start of Abort Loop
5: $\quad\quad\mathbf{z}^* \leftarrow \mathsf{Sample}_Z()$
6: $\quad\quad\mathbf{h}^* \leftarrow \mathsf{Sample}_h()$
7: $\quad\quad\mathbf{w}_1^* = \mathsf{UseHint}(\mathbf{h}^*, \mathbf{A} \cdot \mathbf{z}^*)$
8: $\quad\quad c = \mathcal{H}(\mu, \mathbf{w}_1^*)$
9: $\quad$**end while**
10: $\quad\sigma = (\mathbf{z}, \mathbf{h}, c)$
11: **end procedure**

7 Experimental Validation

7.1 Experimental Setup

Our experiments are performed on the optimized implementations of Kyber and Dilithium, taken from the *pqm4* library, a benchmarking and testing framework for PQC schemes on the ARM Cortex-M4 family of microcontrollers [KRSS19]. Our DUT is the STM32F407VG microcontroller mounted on the STM32F4DISCOVERY board. The implementations are compiled using the `arm-none-eabi-gcc` compiler (with compilation options `-O3 -mthumb -mcpu=cortex-m4 -mfloat-abi=hard -mfpu=fpv4-sp-d16`) and run at a clock frequency of 168 MHz. The DUT contains cache lines for both instruction and data fetched from flash memory, to accelerate code execution and literal access. Both the instruction and data caches are therefore enabled to maximize performance. The communication with the DUT is done using UART.

We rely on Electromagnetic Fault Injection as our attack vector. Our EMFI setup comprises of three main components: (1) a high-voltage pulse generator capable of generating pulses up to 200V (in either polarity) with a very low rise time under 4ns; (2) a hand-crafted electromagnetic probe designed as a simple loop antenna; and (3) a motorised XYZ table to position the probe over the DUT. An optional oscilloscope is used for verification of pulse strength and timing characteristics. A software synchronizes the operation of the DUT and the EMFI setup, with faults injected based on a feedback signal from the DUT. Relay switches are also used for automated power-on reset of the DUT.

7.2 Performing Targeted Fault Injection

For our attack evaluation, we utilize a trigger signal from the DUT to signal the start of the target the NTT to fault. However, an attacker can also utilize EM/power side-channel information to approximately narrow down the time window for fault injection.

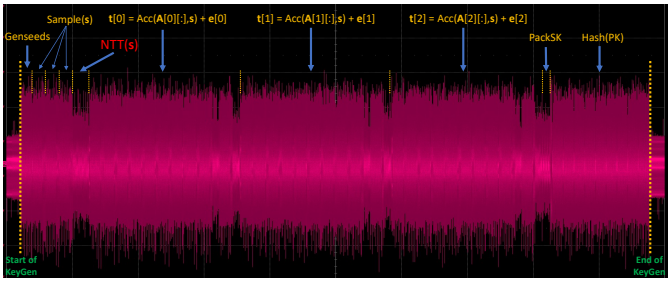
7.2.1 Using Power/EM Analysis for Identification of Time Window

We utilize EM measurements acquired from the same DUT using a near-field EM probe, collected using a Lecroy 610Zi oscilloscope at a sampling rate of 500MSam/sec. The repetitive nature of operations in Module-LWE/LWR based schemes, as well as a preliminary knowledge of the implementation allows us to distinguish different operations. Refer Fig.6(a) for the EM trace from execution of the key generation procedure of Kyber768, where we annotate the trace with names of different operations. Refer Fig.6(b) for a zoomed-in-view of the trace which clearly shows the repeating patterns corresponding to the $k = 3$ NTTs of $\mathbf{s}$. We also confirmed through experiments that a similar technique can be applied to the Kyber's encryption procedure as well as the signing and verification procedures of Dilithium (waveforms are omitted for brevity).

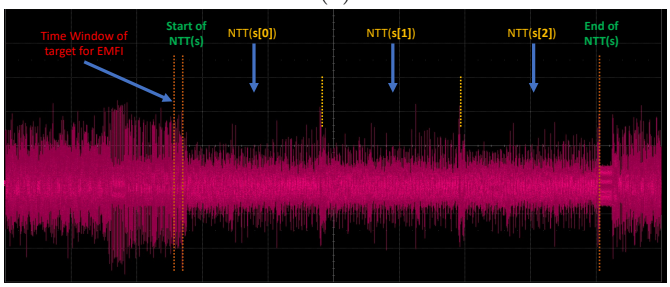
Upon roughly identifying the time window of the target NTTs, the attacker's main target is the twiddle-pointer loading operation that occurs just before the start of the NTT operation. Our EM side-channel analysis allows to narrow the time window to about 100-200 ns for fault injection.

7.2.2 Faulting Multiple NTTs in a Single Execution:

Our proposed attacks barring Sign_Fault_NTT_C and Verification-Bypass on Dilithium, require to fault multiple NTT instances in a single execution. For instance, the Kyber-Key-Recovery attack requires to fault $k = 3$ NTTs of $\mathbf{s}$ in the key generation procedure, which would typically require 3 faults, one in each NTT. However, we observed through practical experiments that a single fault on the first NTT of a given module $\mathbf{s}$ (i.e.) $\mathbf{s}[0]$ propagates to the NTTs on all the other polynomials of $\mathbf{s}$ (i.e.) $\mathbf{s}[i]$ for $i \in [1, k-1]$. The same effect



(a)



(b)

Figure 6: Visual Inspection of EM trace from key generation procedure of Kyber768 on the ARM Cortex-M4 microcontroller (a) Identification of repeating patterns and mapping to different operations (b) zoomed-in-view of trace corresponding to the $k = 3$ NTTs of $\mathbf{s}$

is also observed on Dilithium, when faulting $\mathbf{y} \in R_q^\ell$ with $\ell = 5$ NTTs. Moreover, the fault only propagates to the NTTs of the the same module, while not affecting the NTT over other modules.

We hypothesize that the aforementioned fault propagation behaviour could be due to reuse of the twiddle-pointer for NTTs of the same module. We recall that the data cache to the flash memory is enabled on our DUT. Hence, it is possible that the twiddle-pointer first retrieved from flash memory for NTT of $\mathbf{s}[0]$ is stored within the data cache, and the subsequent NTTs reuse the cached twiddle-pointer, without actually fetching from the flash memory. Thus, faulting the first fetch of the twiddle-pointer from flash memory ensures that a faulty value is also used for the subsequent NTTs of the same module. Thus, all our proposed attacks on both Kyber and Dilithium, require to inject only a single targeted fault in the target computation. This therefore serves as a best case scenario for an attacker, where a single fault is sufficient to fault multiple NTTs of the same module. However, we are only able to provide a hypothesis for our aforementioned observed behavior, as we are unable to fully analyze the effect of the injected faults at the micro-architectural level on our DUT.

7.3 Fault Injection Results

We consider the case of a profiled attacker who can profile the device and obtain the ideal set of fault injection parameters (i.e.) voltage (v), pulse-width (w), delay (d), x-y coordinate of the probe on chip (xy), that yields high repeatability. We refer to a given set of values for the parameters (i.e.) (v_i, w_i, d_i, xy_i) as an *injection instance*. The number of repeated experiments performed at each injection instance is denoted as the *repetition* count.

To obtain injection instances that yield the best fault repeatability, we follow a two-step approach. We first perform a preliminary fault injection campaign, sweeping coarsely over a range of values for all the fault injection parameters, covering the entire area of the chip, and running 5 repetitions at each injection instance. We were able to achieve faults

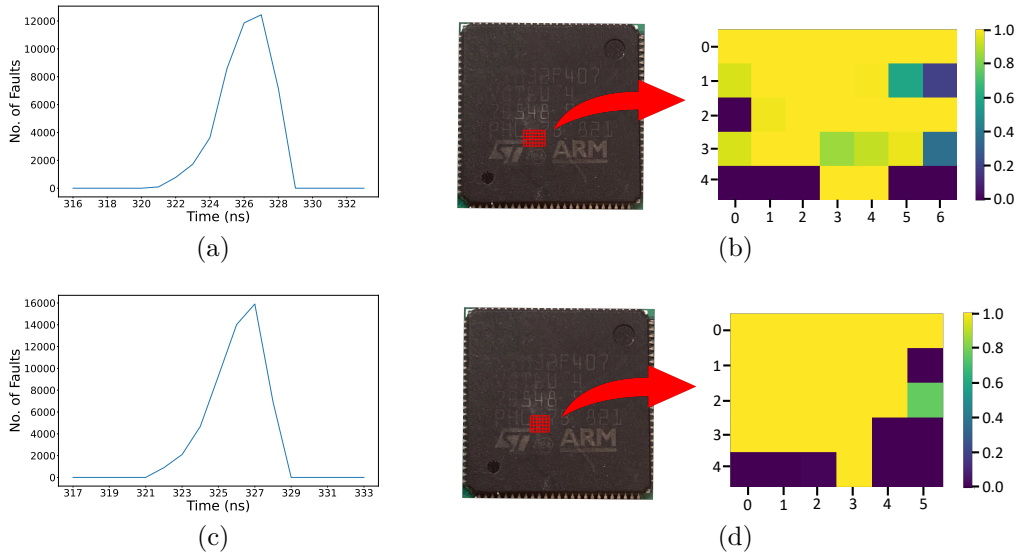


Figure 7: EMFI Results for Kyber-Key-Recovery (a,b) and Kyber-Message-Recovery (c,d) for Kyber768. (a,c) denotes sensitive time window, while (b,d) denotes best fault repetability achievable at different sensitive locations (XY) for the corresponding attacks.

with high repeatability for voltage in the range of 140-170v, and pulse width of 7 nsecs. Based on results from the preliminary campaign, we narrowed down the area for high fault repeatability, and ran a more detailed campaign with 100 repetitions at each selected instance to calculate concrete numbers for fault repeatability. Results from the latter are presented in the following.

7.3.1 Kyber-Key-Recovery

We performed a total of 69300 fault injection experiments (i.e.) 100 experiments each at 693 favourable injection instances, to zeroize the twiddle constants of all the $k = 3$ NTTs of $\mathbf{s}$ in the key generation procedure of Kyber768. Among them, we obtained 46281 successful faults ($\approx 66\%$) and the number of successful faults against the injection delay is shown in Fig.7(a). We observe a narrow time window of about 7 ns in which we can observe a very high number of successful faults. Refer Fig.7(b) for the best fault repeatability achievable (across voltage, pulse width and injection delay) as a function of the xy location of the injection probe on the chip's surface. We can observe that there are several fault injection instances (in a 1 mm $\times$ 1.5 mm area) that yield a high fault repeatability up to 100%. We also tested our key recovery attack on 100 random faulty public keys obtained from one such fault injection instance. We were able to recover the secret key with 100% success rate, while the faulty public keys also resulted in correct key exchanges.

7.3.2 Kyber-Message-Recovery

We performed 64600 fault injections to fault the $k = 3$ NTTs of the ephemeral secret $\mathbf{r}$ of Kyber's encryption procedure, among which we obtained 53844 successful faults ($\approx 83\%$). Refer Fig.7(c)-(d) for the corresponding fault injection results which very closely resembles the results of our Kyber-Key-Recovery attack. We yet again observe very high repeatability of up to 100% at several fault injection instances. We also experimentally verified our message recovery attack on 100 random faulty ciphertexts, which yielded 100% success rate for recovering the message and the corresponding shared secret.

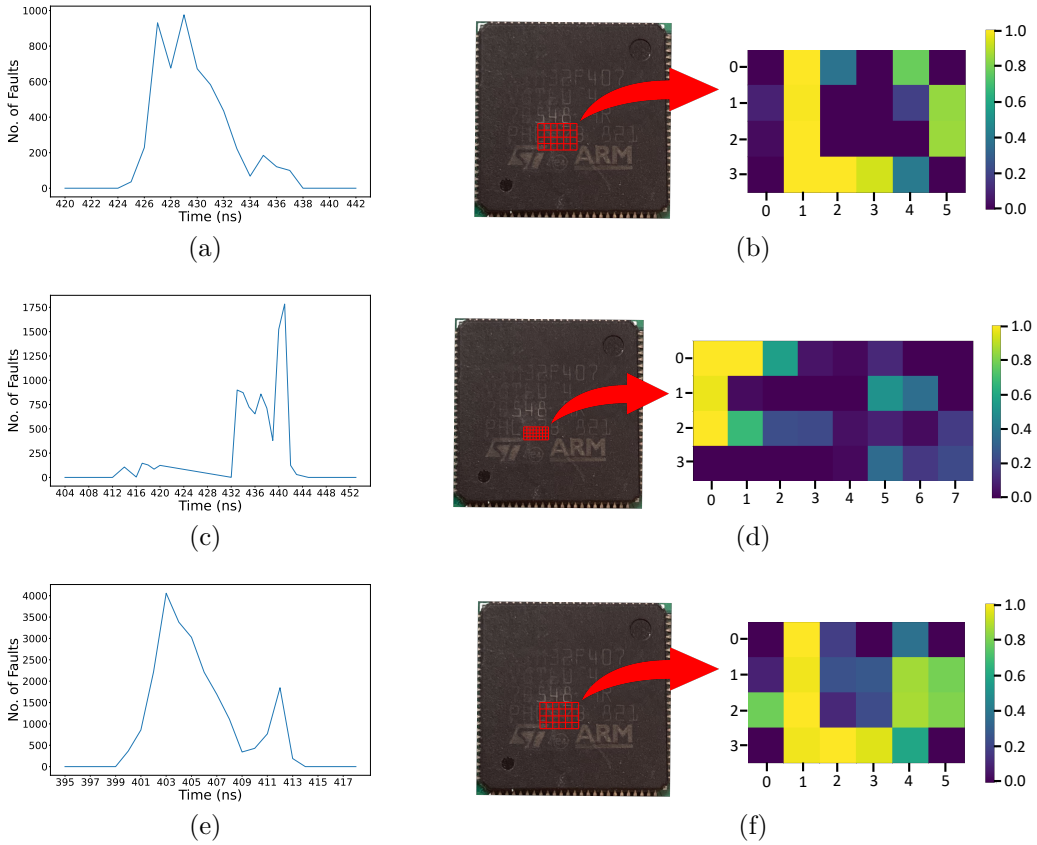


Figure 8: EMFI Results for Sign_Fault_NTT_C on deterministic signing procedure (a,b) and Sign_Fault_NTT_Y on probabilistic signing procedure (c,d) and Verification-Bypass (e,f) for Dilithium3. (a,c,e) denotes Sensitive Time window, while (b,d,f) denotes best fault repetability achievable at different sensitive locations (XY) for the corresponding attacks.

7.3.3 Sign_Fault_NTT_C

We performed a total of 10100 fault injection experiments to fault the NTT of the challenge polynomial c in the signing procedure of deterministic Dilithium. We obtained a total of 5234 successful faults ($\approx 51\%$), all observed within a narrow time window of 13 ns (Refer Fig.8(a)). Refer Fig.8(b) for the cartography of the best achievable fault repeatability (in a 1.5 mm $\times$ 2.5 mm area) on the DUT. This clearly shows several locations that yield high fault repeatability up to 100%. We tested our attack on about 100 random faulty signatures and obtained a 100% success rate for key recovery.

7.3.4 Sign_Fault_NTT_Y

We performed a total of 50300 fault injection experiments to fault all the $\ell = 5$ NTTs of the ephemeral nonce $\mathbf{y}$ in the signing procedure of probabilistic Dilithium. We obtained a total of 9155 successful faults ($\approx 26\%$), all observed within a slightly wider time window of 30 ns (Refer Fig.8(c)). Refer Fig.8(d) for the cartography of the best achievable fault repeatability (in a 0.75 mm $\times$ 2 mm area), which again shows multiple locations that yield high fault repeatability up to 100%. We tested our attack on about 100 random faulty signatures and obtained a 100% success rate for key recovery, while all the faulty signatures successfully passed the verification procedure.

7.3.5 Verification-Bypass

We performed a total of 35000 fault injection experiments the NTT of the challenge polynomial c in the verification procedure. We obtained a total of 22487 successful faults ($\approx 64\%$), all observed within a time window of 23 ns (Refer Fig.8(e)). Refer Fig.8(f) for the best fault repeatability achievable as a function of the location injection probe on the chip's surface (in a 1.5 mm $\times$ 2.5 mm area), which again shows several locations that yield high fault repeatability up to 100%. We also experimentally verified that invalid signatures for attacker's chosen messages were successfully verified with a 100% success rate.

7.3.6 Summary of Results

Thus, for all our targets, we observed between 26%-83% faults, that were successful when performing a detailed fault injection campaign for selected fault injection instances. The existence of yellow spots in Fig.7,8 clearly demonstrates the possibility to achieve high fault repeatability for all of our presented attacks. Once an adversary has identified one such fault injection instance, the attack success rate is 100%.

7.4 Attacking Fault Protected Implementations

In this section, we discuss the applicability of our attacks to protected implementations of Kyber and Dilithium hardened against known fault attacks. Refer to Section 3 for a detailed discussion on known fault attacks and countermeasures/mitigations for both Kyber and Dilithium. For Kyber KEM, we only focus on attacks on the key-generation and encryption/encapsulation procedure. For Dilithium, we focus on attacks on the signing and verification procedure. To the best of our knowledge, we are not aware of publicly available fault protected implementations of Kyber/Dilithium. Thus, we implement the aforementioned countermeasures on the optimized implementations of Kyber and Dilithium from the *pqm4* library and perform our analysis on the same. All the implemented countermeasures can be separately turned on/off based on the user requirements.

7.4.1 Targeting Fault Protected Kyber

We experimentally validated our Kyber-Key-Recovery attack on the key-generation procedure of Kyber KEM protected with the Verify_Nonce_Fault countermeasure. We were able to achieve 100% success rate in key recovery, similar to that of our attack on the unprotected implementation with the same fault injection parameters. The countermeasure checks for repetition of polynomials in the secret module $\mathbf{s}$ and error module $\mathbf{e}$ after the sampling procedure (Line 5,6 in Alg.1), while our attack targets the NTT of $\mathbf{s}$, after countermeasure is executed. Thus, it is trivial to see that our attack can easily bypass the Verify_Nonce_Fault countermeasure.

We also validated our attack on the shuffled NTT implementation proposed by Ravi *et al.* [RPBC20], which involves shuffling the order of operations within the NTT. While there are different variants of the shuffled NTT, we validated our attack on the assembly optimized implementation of the fine-shuffled NTT variant. We henceforth refer to this countermeasure as the Shuffled_NTT countermeasure. As discussed earlier in Section 4.3.1, the shuffling countermeasure is orthogonal to our attack which works by faulting the twiddle factor data. Thus, as expected, our attack also works on the fine-shuffled variant of NTT, and therefore believe that our attack similarly applies to all the other shuffled variants of the NTT. Moreover, it is trivial to see that that the Shuffled_NTT countermeasure can also be bypassed by our attack targeting NTTs in the Dilithium signature scheme.

Table 2: Ability of our proposed attacks to bypass fault countermeasures against known attacks on key-generation and encryption procedure of Kyber KEM

Countermeasure	Attack (KeyGen & Encaps)	
	Kyber-Key-Recovery	Kyber-Message-Recovery
Verify_Nonce_Fault	✓	✓

Similar to our attack on the protected key-generation procedure, our Kyber-Message-Recovery attack is also applicable to the protected encapsulation procedure of Kyber KEM, when hardened with the same countermeasures. Please refer Tab.2 for the summary of applicability of our attacks on the known fault countermeasures for Kyber and Dilithium.

7.4.2 Targeting Fault Protected Dilithium

We experimentally validated our key recovery attacks on the signing procedure of Dilithium hardened with the following three countermeasures: Verify_After_Sign, Verify_Loop_Abort and Verify_Add. We implemented the Sign_Fault_NTT_Y attack, targeting NTT($\mathbf{y}$) (Line 19 in Alg.2) in the probabilistic signing procedure of Dilithium. We were able to achieve 100% success rate in key recovery, similar to that of our attack on the unprotected implementation with the same fault injection parameters.

We recall that this attack generates valid signatures which always pass verification. Thus, this attack can easily bypass the Verify_After_Sign countermeasure. The Verify_Loop_Abort countermeasure only checks against skipping attacks that target the sampling of the nonce $\mathbf{y}$ (Line 19), while our faults are injected in the NTT over $\mathbf{y}$ after the countermeasure is executed. Moreover, Verify_Add countermeasure protects against skipping attacks targeting the final addition operation to generate $\mathbf{z}$ (Line 24), while our attack targets the NTT operation that occurs much earlier (Line 19). Thus, we can see that all existing fault countermeasures for the signing procedure are orthogonal to our attack, and can thus be easily bypassed with our Sign_Fault_NTT_Y attack. Nevertheless, we recall that the Sign_Fault_NTT_Y attack is only possible when NTT($\mathbf{y}$) is used to generate $\mathbf{z}$ as discussed in Section 6.1.2. While we performed the attack on the probabilistic signing procedure, the same attack applies in the same manner to the deterministic signing procedure as well.

Table 3: Ability of our proposed attacks to bypass fault countermeasures against known attacks on signing procedure of Dilithium signature scheme.

Countermeasure	Attack (Sign)	
	Sign_Fault_NTT_C	Sign_Fault_NTT_Y
Deterministic Signing		
Verify_Loop_Abort	✓	✓
Verify_After_Sign	✗	✓
Verify_Add	✓	✓
Probabilistic Signing		
Verify_Loop_Abort	✗	✓
Verify_After_Sign	✗	✓
Verify_Add	✗	✓

We recall that our Sign_Fault_NTT_C attack on the signing procedure of deterministic Dilithium, targets the NTT operation over the challenge polynomial c (Line 23). It results in faulty signatures that are invalid, which always fail verification. Thus, the Verify_After_Sign countermeasure acts as a strong deterrent against the attack. However, our attack can easily bypass Verify_Loop_Abort and Verify_Add countermeasures, in the same manner as that of our Sign_Fault_NTT_Y attack. We do not implement

countermeasures for the verification procedure of Dilithium, as they have not been subjected to practical fault attacks. Please refer Tab.3 for the summary of applicability of our attacks on the known fault countermeasures for Kyber and Dilithium. We believe our study warrants more research towards dedicated fault countermeasures for the NTT, used in post-quantum KEMs and signature schemes.

8 Countermeasures

We have concretely shown through practical experiments that twiddle-pointer vulnerability enables a variety of attacks on both Kyber and Dilithium, while also capable of bypassing existing countermeasures. In this section, we present a succinct discussion on dedicated countermeasures that can mitigate our proposed attacks exploiting the twiddle-pointer vulnerability. We categorize our countermeasures into two types: (1) Implementation-Level and (2) Algorithmic-Level countermeasures.

8.1 Implementation-Level Countermeasures

These countermeasures are designed at the implementation/design level to remove the twiddle-pointer vulnerability, or reduce the ability of the attacker to precisely inject faults during the twiddle-pointer loading operation.

1. **Jitter and Horizontal Noise:** Our attack requires to inject precisely targeted faults to manipulate the loading of twiddle-pointer from flash memory. Thus, introduction of jitter around this target operation has a significant impact of the attack's success rate, depending upon the amount of introduced jitter. While this does not completely prevent the attack, it can serve as an efficient and low-cost mitigation technique.

2. **On-the-fly Computation of Twiddle Factors:** Instead of pre-computing the twiddle constants, one can adopt an on-the-fly approach to compute the twiddle constants for the NTT/INTT, thereby eliminating the twiddle-pointer vulnerability. However, on-the-fly computation of the twiddle constants could impose a heavy performance penalty on the NTT/INTT.

3. **Twiddle Pointer Integrity Checks:** One can also utilize parity checks or dummy registers for redundant loading of the twiddle pointer, to detect any faults on the twiddle pointer value.

8.2 Algorithmic-Level Countermeasures

These countermeasures attempt to detect faults in the twiddle factors through exploitation of the inherent properties of the NTT operation.

1. **Checking Sanity of Twiddle Pointer Array:** We also observe that twiddle factors are nothing but powers of the n^{th} roots of unity (denoted as ω), which satisfy the property that $\omega^n = 1$ and $\omega^{n/2} = q - 1$. Once the twiddle pointer is loaded for the NTT operation, we can check whether the aforementioned arithemtic properties are satisfied. To check whether $\omega^n = 1$, one can pick t twiddle factors at random such that their product is expected to be $\omega^n = 1$. If the comparison passes successfully, only then do we proceed with the NTT operation. For the faulty NTT with zero twiddle factor array, this comparison always fails, thereby providing concrete protection against complete zeroization of twiddle factors.

2. **Computing Entropy of NTT Output:** A closer observation of the faulty NTT output reveals that all of its n coefficients have a fixed value (i.e.) first coefficient

repeating n times. Thus, the faulty NTT output has very little entropy compared to the correct NTT output, which consists of uniformly random coefficients in the range $[0, q]$. Thus, a simple check to test the entropy/distribution of coefficients can be used to detect fault in the NTT.

Thus, a designer can employ a combination of the aforementioned implementation-level and algorithmic-level countermeasures to provide strong protection against our proposed fault attacks on the NTT.

9 Conclusion

In this paper, we have shown that the twiddle-pointer vulnerability enables a variety of attacks on practical implementations of both Kyber and Dilithium. We demonstrate novel key recovery and message recovery attacks on Kyber and key recovery and verification bypass attacks on Dilithium, using Electromagnetic Fault Injection which work with a 100% success rate on optimized implementations of Kyber and Dilithium on the ARM Cortex-M4 microcontroller. We also demonstrate that our attacks are able to bypass known fault countermeasures. Since our attack targets the inherent properties of the NTT, we believe our attacks can be extended to other schemes such as Saber, NTRU and NTRU Prime, which also utilize the NTT for polynomial multiplication. Our work stresses the need for concrete custom countermeasures against fault injection attacks for practical implementations of the NTT, especially in embedded devices.

Acknowledgements

This work was supported in part by National Key R&D Program of China (2020AAA0107700), by National Natural Science Foundation of China (62227805, 62072398), by Alibaba-Zhejiang University Joint Institute of Frontier Technologies, by Leading Innovative and Entrepreneur Team Introduction Program of Zhejiang (2018R01005), and by Research Institute of Cyberspace Governance in Zhejiang University, by National Key Laboratory of Science and Technology on Information System Security (6142111210301), by State Key Laboratory of Mathematical Engineering and Advanced Computing, and by Open Foundation of Henan Key Laboratory of Cyberspace Situation Awareness (HNTS2022001). The authors would also like to acknowledge the financial support received from the Singapore National Research Foundation under the SoCure NRF2018NCR-NCR002-0001 grant (www.green-ic.org/socure) for carrying out this research.

References

[AAC+22] Gorjan Alagic, Daniel Apon, David Cooper, Quynh Dang, Thinh Dang, John Kelsey, Jacob Lichtinger, Carl Miller, Dustin Moody, Rene Peralta, et al. Status report on the third round of the NIST post-quantum cryptography standardization process. Technical report, National Institute of Standards and Technology, 2022.

[AASA+20] Gorjan Alagic, Jacob Alperin-Sheriff, Daniel Apon, David Cooper, Quynh Dang, John Kelsey, Yi-Kai Liu, Carl Miller, Dustin Moody, Rene Peralta, et al. Status report on the second round of the NIST post-quantum cryptography standardization process. *US Department of Commerce, NIST*, 2020.

[ABCG20] Erdem Alkim, Yusuf Alper Bilgin, Murat Cenk, and François Gérard. Cortex-m4 optimizations for R, M LWE schemes. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 336–357, 2020.

[ABD+20] Roberto Avanzi, Joppe W. Bos, Leo Ducas, Eike Kiltz, Tancrede Lepoint, Vadim Lyubashevsky, John Schanck, Peter Schwabe, Gregor Seiler, and Damien Stehlé. CRYSTALS-Kyber (version 3.0): Algorithm specifications and supporting documentation (October 1, 2020). *Submission to the NIST post-quantum project*, 2020.

[ACC+21] Erdem Alkim, Dean Yun-Li Cheng, Chi-Ming Marvin Chung, Hülya Evkan, Leo Wei-Lun Huang, Vincent Hwang, Ching-Lin Trista Li, Ruben Niederhagen, Cheng-Jhih Shih, Julian Wälde, et al. Polynomial multiplication in NTRU prime: Comparison of optimization strategies on Cortex-M4. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 217–238, 2021.

[ACC+22] Amin Abdulrahman, Jiun-Peng Chen, Yu-Jia Chen, Vincent Hwang, Matthias J Kannwischer, and Bo-Yin Yang. Multi-moduli NTTs for Saber on Cortex-M3 and Cortex-M4. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 127–151, 2022.

[ADP18] Martin R Albrecht, Amit Deo, and Kenneth G Paterson. Cold boot attacks on ring and module LWE keys under the NTT. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 173–213, 2018.

[AH21] Daniel Apon and James Howe. Attacks on NIST PQC 3rd Round Candidates, 2021. Invited talk at Real World Crypto 2021, https://iacr.org/submit/files/slides/2021/rwc/rwc2021/22/slides.pdf.

[AHKS22] Amin Abdulrahman, Vincent Hwang, Matthias J Kannwischer, and Daan Sprenkels. Faster Kyber and Dilithium on the Cortex-M4. *Cryptology ePrint Archive*, 2022.

[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 2016, Santa Barbara, CA, USA, August 16, 2016*, pages 63–77. IEEE Computer Society, 2016.

[BKS19] Leon Botros, Matthias J Kannwischer, and Peter Schwabe. Memory-efficient high-speed implementation of Kyber on Cortex-M4. In *International Conference on Cryptology in Africa*, pages 209–228. Springer, 2019.

[BP18] Leon Groot Bruinderink and Peter Pessl. Differential fault attacks on deterministic lattice signatures. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2018(3), 2018. https://eprint.iacr.org/2018/355.pdf.

[CHK+21] Chi-Ming Marvin Chung, Vincent Hwang, Matthias J Kannwischer, Gregor Seiler, Cheng-Jhih Shih, and Bo-Yin Yang. NTT multiplication for NTT-unfriendly rings: New speed records for Saber and NTRU on Cortex-M4 and AVX2. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 159–188, 2021.

[Coo66] SA Cook. On the minimum computation time for multiplication. *Doctoral diss., Harvard U., Cambridge, Mass*, 1, 1966.

[CT65] James Cooley and John Tukey. An algorithm for the machine calculation of complex fourier series. *Mathematics of Computation*, 19(90):297–301, 1965.

[CXS+05] Shuo Chen, Jun Xu, Emre Can Sezer, Prachi Gauriar, and Ravishankar K Iyer. Non-control-data attacks are realistic threats. In *USENIX security symposium*, volume 5, page 146, 2005.

[DDLL13] Léo Ducas, Alain Durmus, Tancrède Lepoint, and Vadim Lyubashevsky. Lattice signatures and bimodal gaussians. In *Advances in Cryptology–CRYPTO 2013*, pages 40–56. Springer, 2013.

[Del22] Jeroen Delvaux. Roulette: A diverse family of feasible fault attacks on masked Kyber. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 637–660, 2022.

[EFGT16] Thomas Espitau, Pierre-Alain Fouque, Benoît Gérard, and Mehdi Tibouchi. Loop abort faults on lattice-based fiat-shamir & hash'n sign signatures. *IACR ePrint Archive*, page 449, 2016.

[FO99] Eiichiro Fujisaki and Tatsuaki Okamoto. Secure integration of asymmetric and symmetric encryption schemes. In *Annual International Cryptology Conference*, pages 537–554. Springer, 1999.

[GKOS18] Tim Güneysu, Markus Krausz, Tobias Oder, and Julian Speith. Evaluation of lattice-based signature schemes in embedded systems. In *2018 25th IEEE International Conference on Electronics, Circuits and Systems (ICECS)*, pages 385–388. IEEE, 2018.

[GKS21] Denisa OC Greconici, Matthias J Kannwischer, and Daan Sprenkels. Compact Dilithium implementations on Cortex-M3 and Cortex-M4. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 1–24, 2021.

[GLP12] Tim Güneysu, Vadim Lyubashevsky, and Thomas Pöppelmann. Practical lattice-based cryptography: A signature scheme for embedded systems. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 530–547. Springer, 2012.

[GS66] W. Morven Gentleman and G. Sande. Fast fourier transforms: for fun and profit. In *American Federation of Information Processing Societies: Proceedings of the AFIPS '66 Fall Joint Computer Conference, November 7-10, 1966, San Francisco, California, USA*, volume 29 of *AFIPS Conference Proceedings*, pages 563–578. AFIPS / ACM / Spartan Books, Washington D.C., 1966.

[HHP+21] Mike Hamburg, Julius Hermelink, Robert Primas, Simona Samardjiska, Thomas Schamberger, Silvan Streit, Emanuele Strieder, and Christine van Vredendaal. Chosen ciphertext k-trace attacks on masked CCA2 secure Kyber. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 88–113, 2021.

[HPP21] Julius Hermelink, Peter Pessl, and Thomas Pöppelmann. Fault-enabled chosen-ciphertext attacks on Kyber. In Avishek Adhikari, Ralf Küsters, and Bart Preneel, editors, *Progress in Cryptology - INDOCRYPT 2021 - 22nd International Conference on Cryptology in India, Jaipur, India, December 12-15, 2021, Proceedings*, volume 13143 of *Lecture Notes in Computer Science*, pages 311–334. Springer, 2021.

[HSA+16] Hong Hu, Shweta Shinde, Sendroiu Adrian, Zheng Leong Chua, Prateek Saxena, and Zhenkai Liang. Data-oriented programming: On the expressiveness of non-control data attacks. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 969–986. IEEE, 2016.

[IMS+22] Saad Islam, Koksal Mus, Richa Singh, Patrick Schaumont, and Berk Sunar. Signature correction attack on Dilithium signature scheme. *arXiv preprint arXiv:2203.00637*, 2022.

[Kar63] Anatolii Karatsuba. Multiplication of multidigit numbers on automata. In *Sov. Phys. Dokl.*, volume 7, pages 595–596, 1963.

[KRSS19] Matthias J Kannwischer, Joost Rijneveld, Peter Schwabe, and Ko Stoffelen. pqm4: Testing and benchmarking NIST PQC on ARM Cortex-M4. In *Second PQC Standardization Conference: University of California, Santa Barbara and co-located with Crypto 2019*, pages 1–22, 2019.

[LDK+17] Vadim Lyubashevsky, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Peter Schwabe, Gregor Seiler, Damien Stehlé, and Shi Bai. Crystals-Dilithium. *Submission to the NIST Post-Quantum Cryptography Standardization [NIST]*, 2017.

[LNW+19] Hans Liljestrand, Thomas Nyman, Kui Wang, Carlos Chinea Perez, Jan-Erik Ekberg, and N Asokan. PAC it up: Towards pointer integrity using ARM pointer authentication. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 177–194, 2019.

[Lyu09] Vadim Lyubashevsky. Fiat-shamir with aborts: Applications to lattice and factoring-based signatures. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 598–616. Springer, 2009.

[MBD+19] Alexandre Menu, Shivam Bhasin, Jean-Max Dutertre, Jean-Baptiste Rigaud, and Jean-Luc Danger. Precise spatio-temporal electromagnetic fault injections on data transfers. In *2019 Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*, pages 1–8. IEEE, 2019.

[MBM+22] Catinca Mujdei, Arthur Beckers, Jose Maria Bermudo Mera, Angshuman Karmakar, Lennert Wouters, and Ingrid Verbauwhede. Side-channel analysis of lattice-based post-quantum cryptography: exploiting polynomial multiplication. *Cryptology ePrint Archive*, 2022.

[POG15] Thomas Pöppelmann, Tobias Oder, and Tim Güneysu. High-performance ideal lattice-based cryptography on 8-bit ATxmega microcontrollers. In *Progress in Cryptology - LATINCRYPT 2015 - 4th International Conference on Cryptology and Information Security in Latin America, Guadalajara, Mexico, August 23-26, 2015, Proceedings*, pages 346–365, 2015.

[PP19] Peter Pessl and Robert Primas. More practical single-trace attacks on the number theoretic transform. In *International Conference on Cryptology and Information Security in Latin America*, pages 130–149. Springer, 2019.

[PP21] Peter Pessl and Lukas Prokop. Fault attacks on CCA-secure lattice KEMs. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 37–60, 2021.

[PPM17a] Robert Primas, Peter Pessl, and Stefan Mangard. Single-trace side-channel attacks on masked lattice-based encryption. In *International Conference on Cryptographic Hardware and Embedded Systems*, pages 513–533. Springer, 2017.

[PPM17b] Robert Primas, Peter Pessl, and Stefan Mangard. Single-trace side-channel attacks on masked lattice-based encryption. In *International Conference on Cryptographic Hardware and Embedded Systems*, pages 513–533. Springer, 2017.

[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*, pages 427–440, 2019.

[RPBC20] Prasanna Ravi, Romain Poussier, Shivam Bhasin, and Anupam Chattopadhyay. On configurable SCA countermeasures against single trace attacks for the NTT. In *International Conference on Security, Privacy, and Applied Cryptography Engineering*, pages 123–146. Springer, 2020.

[RR21] Prasanna Ravi and Sujoy Sinha Roy. Side-channel analysis of lattice-based PQC candidates. *Round 3 Seminars, NIST Post Quantum Cryptography*, 2021.

[RRB+19] Prasanna Ravi, Debapriya Basu Roy, Shivam Bhasin, Anupam Chattopadhyay, and Debdeep Mukhopadhyay. Number "not used" once-practical fault attack on pqm4 implementations of NIST candidates. In *International Workshop on Constructive Side-Channel Analysis and Secure Design*, pages 232–250. Springer, 2019.

[RVM+14] Sujoy Sinha Roy, Frederik Vercauteren, Nele Mentens, Donald Donglong Chen, and Ingrid Verbauwhede. Compact ring-LWE cryptoprocessor. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 371–391. Springer, 2014.

[SBH+22] Hadi Soleimany, Nasour Bagheri, Hosein Hadipour, Prasanna Ravi, Shivam Bhasin, and Sara Mansouri. Practical multiple persistent faults analysis. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 367–390, 2022.

[Sha07] Hovav Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *Proceedings of the 14th ACM conference on Computer and communications security*, pages 552–561, 2007.

[SMD+13] Kevin Z Snow, Fabian Monrose, Lucas Davi, Alexandra Dmitrienko, Christopher Liebchen, and Ahmad-Reza Sadeghi. Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization. In *2013 IEEE Symposium on Security and Privacy*, pages 574–588. IEEE, 2013.

[Too63] Andrei L Toom. The complexity of a scheme of functional elements realizing the multiplication of integers. In *Soviet Mathematics Doklady*, volume 3, pages 714–716, 1963.

[XIU+21] Keita Xagawa, Akira Ito, Rei Ueno, Junko Takahashi, and Naofumi Homma. Fault-injection attacks against NIST's post-quantum cryptography round 3 KEM candidates. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 33–61. Springer, 2021.

[YPK+22] Sungbae Yoo, Jinbum Park, Seolheui Kim, Yeji Kim, and Taesoo Kim. In-Kernel Control-Flow integrity on commodity OSes using ARM pointer authentication. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 89–106, 2022.