Side-channel and Fault-injection attacks over Lattice-based Post-quantum Schemes (Kyber, Dilithium): Survey and New Results

PRASANNA RAVI* and ANUPAM CHATTOPADHYAY†, Temasek Labs, Nanyang Technological University, Singapore and School of Computer Science and Engineering, Nanyang Technological University, Singapore

JAN PIETER D'ANVERS‡, imec-COSIC, KU Leuven, Belgium

ANUBHAB BAKSI§, Temasek Labs, Nanyang Technological University, Singapore

In this work, we present a systematic study of Side-Channel Attacks (SCA) and Fault Injection Attacks (FIA) on structured lattice-based schemes, with a focus on Kyber Key Encapsulation Mechanism (KEM) and Dilithium signature scheme, which are leading candidates in the NIST standardization process for Post-Quantum Cryptography (PQC). Through our study, we attempt to understand the underlying similarities and differences between the existing attacks, while classifying them into different categories. Given the wide variety of reported attacks, simultaneous protection against all the attacks requires to implement customized protections/countermeasures for both Kyber and Dilithium. We therefore present a range of customized countermeasures, capable of providing defenses/mitigations against existing SCA/FIA, and incorporate several SCA and FIA countermeasures within a single design of Kyber and Dilithium. Among the several countermeasures discussed in this work, we present novel countermeasures that offer simultaneous protection against several SCA and FIA-based chosen-ciphertext attacks for Kyber KEM. We implement the presented countermeasures within the well-known *pqm4* library for the ARM Cortex-M4 based microcontroller. Our performance evaluation reveals that the presented custom countermeasures incur reasonable performance overheads, on the ARM Cortex-M4 microcontroller. We therefore believe our work argues for the usage of custom countermeasures within real-world implementations of lattice-based schemes, either in a standalone manner or as reinforcements to generic countermeasures such as masking.

CCS Concepts: • **Security and privacy → Side-channel analysis and countermeasures**.

Additional Key Words and Phrases: Lattice-based Cryptography, Side-Channel Attacks, Fault-Injection Attacks, Kyber, Dilithium

ACM Reference Format:

Prasanna Ravi, Anupam Chattopadhyay, Jan Pieter D'Anvers, and Anubhab Baksi. 2022. Side-channel and Fault-injection attacks over Lattice-based Post-quantum Schemes (Kyber, Dilithium): Survey and New Results. 1, 1 (November 2022), 50 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

In 2016, the National Institute for Standards and Technology (NIST) initialized a global level standardization process for *quantum attack* resistant public-key cryptographic schemes [2], which is otherwise known as *Post-Quantum Cryptography (PQC)*. Very recently in 2022, after three rounds of evaluation, NIST announced the first standards for PQC

Authors' addresses: Prasanna Ravi, prasanna.ravi@ntu.edu.sg; Anupam Chattopadhyay, anupam@ntu.edu.sg, Temasek Labs, Nanyang Technological University, Singapore and School of Computer Science and Engineering, Nanyang Technological University, Singapore; Jan Pieter D'Anvers, janpieter.danvers@esat.kuleuven.be, janpieter.danvers@esat.kuleuven.be, imec-COSIC, KU Leuven, Belgium; Anubhab Baksi, anubhab.baksi@ntu.edu.sg, anubhab.baksi@ntu.edu.sg, Temasek Labs, Nanyang Technological University, Singapore.

Manuscript submitted to ACM

in the category of Public Key Encryption (PKE), Key Encapsulation Mechanisms (KEM) and Digital Signatures (DS) [3]. Theoretical post-quantum security guarantees and implementation performance on different HW/SW platforms served as the primary criteria for selection in the initial rounds of the NIST standardization process. However, resistance against side-channel attacks (SCA) and fault injection attacks (FIA) as well as the cost of implementing protections against SCA and FIA, also emerged as a very important criterion towards the latter part of the standardization process. This is especially true when it comes to comparing schemes with tightly matched security and efficiency [5]. In [2, Sections 3.4 and 2.2.3] NIST states that it *encourages additional research regarding side-channel analysis* of the finalist candidates and *hopes to collect more information about the costs of implementing these algorithms in a way that provides resistance to such attacks*.

Three out of the seven finalist candidates derive their hardness from variants of the well-known Learning With Error (LWE) and Learning With Rounding (LWR) problems. In this respect, Kyber [6] and Dilithium [25], which are based on the Module-LWE problem, were selected as the first standards for KEMs and signature schemes respectively. Moreover, these schemes have received considerable attention with several works demonstrating practical attacks [11, 53, 54, 65], particularly on embedded targets. These attacks have been realized using a wide range of attack vectors such as power and Electromagnetic Emanation (EM) for SCA and voltage/clock glitching and EM for FIA. The proposed attacks have been quite diverse in nature, in terms of the targeted operation, method of constructing queries to the target device, as well as the mathematical approach for key recovery.

There are existing works such as [46, 60] that have provided a good overview of existing implementations of PQC. However, a similar overview that covers recent developments in the field of SCA and FIA of PQC, and in particular, lattice-based schemes is missing. This is especially important, given the ever-growing list of attacks proposed on practical embedded implementations of lattice-based schemes. The type of attack that is applicable to a given instance of Kyber or Dilithium, depends upon a variety of factors such as the target procedure, target operation within the procedure, attack vector, operating mode of the scheme, experimental setup, and many more. Thus, it is important for a designer to understand the applicability of different attacks for his/her use case. This is especially important if a designer is tasked with choosing suitable countermeasures to provide concrete protection against SCA and FIA.

There has also been significant interest in the cryptographic community towards the development of SCA and FIA countermeasures for lattice-based schemes. They can be broadly classified into two categories - (1) Generic and (2) Custom. Generic countermeasures attempt to provide concrete security guarantees agnostic to the attack strategy, while custom countermeasures are those that offer protection against specific targeted attacks. With respect to SCA, there have been several works that have proposed generic masking strategies for lattice-based schemes [10, 13, 50]. However, one can observe several shortcomings with respect to adopting generic countermeasures such as masking. Firstly, practical attacks have been demonstrated over masked implementations of lattice-based schemes [47], and non-trivial flaws in theoretically secure masking schemes have also been exploited for key-recovery [11]. Secondly, masking has been shown to result in significant performance overheads for both lattice-based KEMs as well as signature schemes, especially on embedded software platforms [13, 33, 44].

In this respect, the contribution of our work is as follows:

(1) We present the first systematic study of SCA/FIA mounted on lattice-based schemes, with the main focus on two leading candidates based on variants of the LWE problem - Kyber KEM [6] and Dilithium signature scheme [25]. Through our study, we attempt to understand the underlying similarities and differences between the existing

attacks, while classifying them into different categories. We also discuss appropriate countermeasures for every attack discussed in this work.

(2) While there are proposals for countermeasures for existing SCA and FIA on Kyber and Dilithium, we are not aware of a concrete implementation that incorporates multiple countermeasures into a single design. Moreover, there are existing attacks for which, countermeasures are unknown or not clear. We also, therefore, propose and implement novel countermeasures for some attacks in this work. In particular, we implement and evaluate novel countermeasures that offer simultaneous protection against several SCA and FIA-based chosen-ciphertext attacks for Kyber KEM.

(3) We also implement all the presented countermeasures in this work, within two well-known public software libraries for PQC - (1) *pqm4* library for the ARM Cortex-M4 based microcontroller [38]. Our performance evaluation reveals that the presented custom countermeasures incur reasonable performance overheads, on both the evaluated embedded platforms. We therefore believe our work argues for the usage of custom countermeasures within real-world implementations of lattice-based schemes, either in a standalone manner or as reinforcements to generic countermeasures such as masking.

Organization of the Paper. In Section 2, we provide a generic description of Kyber and Dilithium. In Section 3 and 4, we describe the known side-channel attacks and fault attacks respectively, along with appropriate countermeasures applicable to Kyber KEM. In Section 5 and 6, we describe the known fault-injection attacks and side-channel attacks respectively, along with appropriate countermeasures applicable to Dilithium signature scheme. In Section 7, we demonstrate performance evaluation of all the different countermeasures against SCA and FIA, for both Kyber and Dilithium. In Section 8, we provide concluding remarks for the paper.

Availability of Software. We will publish related software used for this paper in public domain upon acceptance of this work.

2 BACKGROUND

2.1 Notations

Elements in the integer ring $\mathbb{Z}_q$ are denoted by regular font letters viz. $a, b \in \mathbb{Z}_q$, where q is a prime. The i^{th} bit in an element $x \in \mathbb{Z}_q$ is denoted as x_i. Vectors and matrices of integers in $\mathbb{Z}_q$ (i.e.) $\mathbb{Z}_q^k$ and $\mathbb{Z}_q^{k\times\ell}$ are denoted in bold upper case letters. The polynomial ring $\mathbb{Z}_q(x)/\phi(x)$ is denoted as R_q where $\phi(x) = (x^n + 1)$ is its reduction polynomial. We denote $\mathbf{r} \in R_q^{k\times\ell}$ as a *module* of dimension $k \times \ell$. Polynomials in R_q and vectors of polynomials in R_q^k are denoted in bold lowercase letters. Matrices of integers of polynomials (i.e.) $R_q^{k\times\ell}$ are denoted in bold upper case letters. The i^{th} coefficient of a polynomial $\mathbf{a} \in R_q$ is denoted as $\mathbf{a}[i]$ and the i^{th} polynomial of a given module $\mathbf{x} \in R_q^k$ as $\mathbf{x}_i$. Multiplication of two polynomials $\mathbf{a}$ and $\mathbf{b}$ in the ring R_q is denoted as $\mathbf{c} = \mathbf{a} \cdot \mathbf{b} \in R_q$ or $\mathbf{a} \times \mathbf{b} \in R_q$. Byte arrays of length n are denoted as $\mathcal{B}^n$. The i^{th} byte in a byte array $x \in \mathcal{B}^*$ is denoted as $x[i]$. Pointwise/Coefficient-wise multiplication of two polynomials $(a, b) \in R_q$ is denoted as $c = a \circ b \in R_q$. 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 throughout the paper. 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 The Learning With Errors Problem [67]

The hardness of both Kyber and Dilithium are based on variants of the well-known Learning With Errors (LWE) problem. The central component of the LWE problem is the LWE instance.

Definition 2.1 (LWE Instance). *For a given dimension $n \geq 1$, elements in $\mathbb{Z}_q$ with $q > 2$ and a Gaussian error distribution $\mathcal{D}_\sigma(\cdot)$, an LWE instance is defined as the ordered pair $(\mathbf{A}, T) \in \mathbb{Z}_q^n \times \mathbb{Z}_q$ where $\mathbf{A} \leftarrow \mathcal{U}(\mathbb{Z}_q^n)$ and $T = \mathbf{A} \cdot \mathrm{S} + E$ with $\mathrm{S} \leftarrow \mathcal{D}_\sigma(\mathbb{Z}_q^n)$ and $E \leftarrow \mathcal{D}_\sigma(\mathbb{Z}_q)$.*

Given an LWE instance, one can define two variants of the LWE problem - (1) *Search LWE* problem - Given polynomially many LWE instances $(\mathbf{A}, T) \in (\mathbf{Z}_q^n, \mathbf{Z}_q)$, solve for $\mathrm{S} \in \mathbb{Z}_q^n$ and (2) *Decisional LWE* - Given many random instances belonging to either valid LWE instances $(\mathbf{A}, T) \in (\mathbf{Z}_q^n, \mathbf{Z}_q)$ or uniformly random instances drawn from $\mathcal{U}(\mathbb{Z}_q^n \times \mathbb{Z}_q)$, distinguish the valid LWE instances from randomly selected ones.

Cryptographic schemes built upon the standard LWE problem suffered from quadratic key sizes and computational times in the dimension n of the lattice (i.e.) $O(n^2)$ [67]. Thus, most of the lattice-based schemes, especially those in the NIST standardization process are based on *algebraically structured* variants of the standard LWE and LWR problem known as the Ring/Module-LWE (RLWE/MLWE) problems respectively. The ring variant of the LWE problem (RLWE) [42] deals with computation over polynomials in polynomial rings $R_q = \mathbb{Z}_q[x]/(x^n + 1)$ with $\mathbf{s}, \mathbf{e} \leftarrow \mathcal{D}_\sigma(R_q)$ such that the corresponding RLWE instance is defined as $(\mathbf{a}, \mathbf{t} = \mathbf{a} \times \mathbf{s} + \mathbf{e}) \in (R_q \times R_q)$. The module variant deals with computations over vectors/matrices of polynomials in $R_q^{k_1 \times k_2}$ with $(k_1, k_2) > 1$. With $\mathbf{A} \leftarrow \mathcal{U}(R_q^{k_1 \times k_2})$ and $\mathbf{s} \leftarrow \mathcal{D}_\sigma(R_q^{k_2})$ and $\mathbf{e} \leftarrow \mathcal{D}_\sigma(R_q^{k_1})$. the corresponding MLWE instance is defined as $(\mathbf{a}, \mathbf{t} = \mathbf{a} \times \mathbf{s} + \mathbf{e}) \in (R_q^{k_1 \times k_2}, R_q^{k_2})$.

2.3 Number Theoretic Transform (NTT) based Polynomial Multiplication

Polynomial multiplication is one of the most computationally intensive operations in structured lattice-based schemes such as Kyber and Dilithium. Both Kyber and Dilithium are designed with parameters that allow the use of the well-known Number Theoretic Transform (NTT) for polynomial multiplication. The NTT is simply a bijective mapping for a polynomial $\mathbf{p} \in R_q$ from a *normal* domain into an alternative representation $\hat{\mathbf{p}} \in R_q$ in the *NTT domain* as follows:

$$\hat{\mathbf{p}}[j] = \sum_{i=0}^{n-1} \mathbf{p}[i] \cdot \omega^{i \cdot j} \tag{1}$$

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 $\hat{\mathbf{p}}$ in the NTT domain back to $\mathbf{p}$ in the normal domain. The use of NTT requires the presence of either the n^{th} root of unity (ω) or $2n^{\text{th}}$ root of unity (ψ) in $\mathbb{Z}_q$ ($\psi^2 = \omega$), 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:

$$\mathbf{c} = \mathrm{INTT}(\mathrm{NTT}(\mathbf{a}) \circ \mathrm{NTT}(\mathbf{b})). \tag{2}$$

The NTT over an n point sequence is performed using the well-known *butterfly* network, which operates over $\log_2(n)$ stages. Refer to the algorithmic specification document of Kyber and Dilithium, on more information about the NTT used in the respective schemes [6, 25].

2.4 Kyber

2.4.1 Algorithmic Description. Kyber is a chosen-ciphertext secure (CCA-secure) KEM based on the Module-LWE problem that has been selected for standardization of PQC-based KEMs, owing to its strong theoretical security guarantees and implementation performance [6]. Computations are performed over modules in dimension $(k \times k)$ (i.e) $R_q^{k\times k}$. Kyber provides three security levels with Kyber512 (NIST Security Level 1), Kyber768 (Level 3) and Kyber1024 (Level 5) with $k = 2, 3$ and 4 respectively. Kyber operates over the anti-cyclic ring R_q with a prime modulus $q = 3329$ and degree $n = 256$, which allows the use of Number Theoretic Transform (NTT) for polynomial multiplication. The CCA-secure Kyber contains in its core, a chosen-plaintext secure encryption scheme of Kyber (IND-CPA secure Kyber PKE), which is based on the well-known framework of the LPR encryption scheme [42].

Refer to Algorithm 1 for a simplified description of the key-generation, encryption and decryption procedures of IND-CPA secure Kyber PKE. 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 $u' \in \mathbb{Z}_q$.

Security and Correctness of IND-CPA Secure Kyber PKE

The key-generation procedure of Kyber PKE simply involves the generation of an LWE instance $(\mathbf{A}, \mathbf{t}) \in (R_q^{k\times k} \times R_q^k)$ where $\mathbf{t} = \mathbf{A} \cdot \mathbf{s} + \mathbf{e}$ (Line 9 in Alg.1). The module $\mathbf{A}$ is sampled from a uniform distribution (Line 4), while the secret $\mathbf{s}$ and errors $\mathbf{e}$ are sampled from a centered binomial distribution (CBD, Lines 5-6). Given that NTT is used for polynomial multiplication, the public key and secret key are directly represented in the NTT domain (Line 10). The LWE instance $(\mathbf{A}, \mathbf{t})$ is the public key, while the secret $\mathbf{s}$ forms the secret key.

The encryption procedure involves generation of two LWE instances $(\mathbf{u}, \mathbf{v}) \in (R_q^k \times R_q)$. The first LWE instance is generated as $\mathbf{u} = \mathbf{A}^T \cdot \mathbf{r} + \mathbf{e_1}$ (Line 18) and the second LWE instance is generated as $\mathbf{v}_p = \mathbf{t}^T \cdot \mathbf{r} + \mathbf{e}_2$ (Line 19). The message to be encrypted (i.e.) $m \in \mathcal{B}^*$ is encoded into a message polynomial $\mathbf{m} \in R_q$, one bit at a time. This is done using the Encode function in the following manner (Line 20). If a message bit $m_i = 1$, then the corresponding coefficient $\mathbf{m}[i] = \lceil q/2 \rfloor$, else $\mathbf{m}[i] = 0$ otherwise . Then, this message polynomial $\mathbf{m}$ is additively hidden within $\mathbf{v}_p$ as $\mathbf{v} = \mathbf{v}_p + \mathbf{m}$ (Line 20). Subsequently, the coefficients of $\mathbf{u}$ and $\mathbf{v}$ are lossily compressed to varying degrees (i.e.) d_1 and d_2 bits respectively using the Compress function, and the compressed versions of $\mathbf{u}, \mathbf{v}$ form the ciphertext ct (Line 21).

The decryption procedure lossily extract the polynomials $\mathbf{u}'$ and $\mathbf{v}'$ from the ciphertext ct with $\Delta\mathbf{u} = (\mathbf{u}' - \mathbf{u})$ and $\Delta\mathbf{v} = (\mathbf{v}' - \mathbf{v})$ (Line 24). Subsequently, the decryption procedure computes $\mathbf{m}' = \mathbf{v}' - \mathbf{u}' \cdot \mathbf{s}$ (Lines 25-27), which is nothing but an approximation of the message polynomial $\mathbf{m}$ (i.e.) $\mathbf{m}'$, which is given as follows:

$$\begin{aligned}
\mathbf{m}' &= \mathbf{v}' - \mathbf{s}^T \cdot \mathbf{u}' \\
&= (\mathbf{v} + \Delta\mathbf{v}) - \mathbf{s}^T \cdot (\mathbf{u} + \Delta\mathbf{u}) \\
&= (\mathbf{t}^T \cdot \mathbf{r} + \mathbf{e}_2 + \mathsf{Encode}(m) + \Delta\mathbf{v}) - \mathbf{s}^T \cdot (\mathbf{A}^T \cdot \mathbf{r} + \mathbf{e}_1 + \Delta\mathbf{u}) \\
&= \mathsf{Encode}(m) + (\mathbf{e}^T \cdot \mathbf{r} + \mathbf{e}_2 + \mathbf{s}^T \cdot \mathbf{e}_1 + \mathbf{s}^T \cdot \Delta\mathbf{u} + \Delta\mathbf{v}) \\
&= \mathsf{Encode}(m) + \mathbf{d}
\end{aligned} \tag{3}$$

where $\mathbf{d} = (\mathbf{e}^T \cdot \mathbf{r} + \mathbf{e}_2 + \mathbf{s}^T \cdot \mathbf{e}_1^T + \mathbf{s}^T \cdot \Delta\mathbf{u} + \Delta\mathbf{v})$ is the noise component in $\mathbf{m}'$, which is also linearly dependent on the secret and error $(\mathbf{s}, \mathbf{e})$ of the public-private key pair. The approximate message polynomial $\mathbf{m}'$ is decoded into the message $m' \in \mathcal{B}^*$ one bit at a time in the following manner: If a given message coefficient $\mathbf{m}[i]$ is in the range

Algorithm 1: CPA Secure Kyber PKE (Simplified)

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

12: **procedure** CPA.Encrypt($pk, m \in \{0,1\}^{256}, seed_R \in \{0,1\}^{256}$)
13: $\hat{\mathbf{A}} \in R_q^{k \times k} \leftarrow \mathsf{Expand}(seed_A)$
14: $\mathbf{r} \in R_q^k \leftarrow \mathsf{Sample}_B(seed_R, coins_0)$ ▷ Sample $\mathbf{r}$ using $(Seed_R, coins_0)$
15: $\mathbf{e_1} \in R_q^k \leftarrow \mathsf{Sample}_B(seed_R, coins_1)$ ▷ Sample $\mathbf{e_1}$ using $(Seed_R, coins_1)$
16: $\mathbf{e_2} \in R_q^k \leftarrow \mathsf{Sample}_B(seed_R, coins_2)$ ▷ Sample $\mathbf{e_2}$ using $(Seed_R, coins_2)$
17: $\hat{\mathbf{r}} \in R_q^k \leftarrow \mathsf{NTT}(\mathbf{r})$ ▷ $\mathsf{NTT}(\mathbf{r})$
18: $\mathbf{u} \in R_q^k \leftarrow \mathsf{INTT}(\mathbf{A}^T \circ \hat{\mathbf{r}}) + \mathbf{e_1}$ ▷ $\mathbf{u} = \mathbf{A}^T \cdot \mathbf{r} + \mathbf{e_1}$
19: $\mathbf{v}_p \in R_q \leftarrow \mathsf{INTT}(\hat{t}^T \circ \hat{\mathbf{r}}) + \mathbf{e_2}$ ▷ $\mathbf{v}_p = \mathbf{t}^T \cdot \mathbf{r} + \mathbf{e_2}$
20: $\mathbf{v} = \mathbf{v}_p + \mathsf{Encode}(m)$
21: Return $ct = \mathsf{Compress}(\mathbf{u}, d_1), \mathsf{Compress}(\mathbf{v}, d_2)$
22: **end procedure**

23: **procedure** CPA.Decrypt(sk, ct)
24: $\mathbf{u}' \in R_q^k = \mathsf{Decompress}(\mathbf{u}, d_1)$; $\mathbf{v}' \in R_q^k = \mathsf{Decompress}(\mathbf{v}, d_2)$
25: $\hat{\mathbf{u}}' = \mathsf{NTT}(\mathbf{u}')$
26: $\hat{g}' = \hat{\mathbf{u}}' \circ \hat{\mathbf{s}}$
27: $\mathbf{m}' \in R_q = \mathbf{v}' - \mathsf{INTT}(\hat{g}')$ ▷ $\mathbf{m}' = \mathbf{v}' - \mathbf{u}' \cdot \mathbf{s}$
28: $m' \in \mathcal{B}^* = \mathsf{Decode}(\mathbf{m}')$
29: Return m'
30: **end procedure**

$[q/4, 3q/4]$, then $m_i = 1$, else $m_i = 0$ otherwise (Line 28). This is computed using a specialized decoding routine, which is sketched in the code snippet shown in Fig.1. It takes as input the message polynomial $\mathbf{m}$ and decodes the coefficients, one at a time into corresponding bits in the 32-byte message array m.

```
uint16_t t = (((m->coeffs[8*i+j] << 1) + KYBER_Q/2) / KYBER_Q) & 1;
m[i] |= t << j;
```

Fig. 1. Message Decoding Routine in Kyber KEM, which converts the message polynomial $\mathbf{m} \in R_q$ into a 32-byte message array m, where i denotes the byte location and j denotes the bit location within a given byte.

As long as the absolute value of all the coefficients of the noise $\mathbf{d}$ are less than $q/4$ (i.e.) $\ell_\infty(\mathbf{d}) < q/4$, the message polynomial $\mathbf{m}'$ is decoded to the correct message m (i.e.) $m' = m$. The parameters of the scheme are chosen so as to attain a negligible decryption failure probability. For recommended parameters of Kyber, the decryption failure probability is $\approx 2^{-164}$. While we have only presented a simplified description of Kyber PKE, a more detailed description can be found in [6].

2.4.2 Security Against Chosen-Ciphertext Attacks. The aforementioned PKE is only secure against chosen-plaintext attacks (IND-CPA security) and thus is not secure against chosen-ciphertext attacks. These attacks typically work by querying the decryption procedure with malicious and invalid ciphertexts, and obtaining information about the corresponding decrypted message m'. This information about m' for malicious and invalid ciphertexts can be used to recover the complete secret key.

The CPA secure Kyber PKE is converted into a CCA secure KEM using the well-known Fujisaki-Okamoto transformation [27]. It utilizes a pair of hash functions $\mathcal{H}$ and $\mathcal{G}$ and a key-derivation function KDF, and forms a wrapper around the encryption and decryption procedures, resulting in encapsulation and decapsulation procedures of a CCA secure KEM (Refer Alg.2).

Algorithm 2: FO transform of a CPA-secure Kyber PKE into a CCA-secure Kyber KEM

1: **procedure** CCA.KeyGen
2: $z \leftarrow \{0,1\}^{256}$
3: $(pk, sk') \leftarrow$ CPA.KeyGen()
4: $sk = (sk' \| \mathcal{H}(pk) \| z)$
5: Return (pk, sk)
6: **end procedure**

7: **procedure** CCA.Encaps(pk)
8: $m \leftarrow \{0,1\}^{256}$
9: $m = \mathcal{H}(m)$
10: $(\bar{K}, r) = \mathcal{G}(m \| \mathcal{H}(pk))$ ▷ Generation of pre-key $\bar{K}$
11: $ct =$ CPA.Encrypt(pk, m, r) ▷ Encryption of message m using public key pk
12: $K = \text{KDF}(\bar{K} \| \mathcal{H}(c))$ ▷ Generation of session key
13: Return (ct, K)
14: **end procedure**

15: **procedure** CCA.Decaps(sk, ct)
16: $(pk, \mathcal{H}(pk), z) \leftarrow$ UnpackSK(sk)
17: $m' =$ CPA.Decrypt(sk, ct) ▷ Decryption of ciphertext into message
18: $(\bar{K}', r') = \mathcal{G}(m', \mathcal{H}(pk))$ ▷ Generation of pre-key $\bar{K}$
19: $T = \bar{K}'$
20: $ct_R =$ CPA.Encrypt(pk, m', r') ▷ Re-Encryption of decrypted message
21: **if** (CompareCT$(ct_R, ct) == 0$) **then** ▷ Ciphertext Comparison
22: $T = z$ ▷ Ciphertext Comparison Failure
23: **end if**
24: Return $K = \text{KDF}(T \| \mathcal{H}(ct')$ ▷ Generation of session key
25: **end procedure**

In theory, the FO transform helps protect the decapsulation procedure of KEMs against chosen-ciphertext attacks in the following manner. The message m' obtained after decryption of the received ciphertext ct (Line 17) is hashed with the public key to generate a pre-shared secret $\bar{K}'$ and a seed r (Line 18). The message m' along with the seed r is then fed into a re-encryption procedure to recompute the ciphertext as ct' (Line 20). A subsequent comparison of ct' with the received ciphertext ct helps evaluate the validity of ct (Line 21). For a valid ciphertext, $ct = ct'$ with a very high probability, and as a result, a valid shared secret K dependent upon the pre-shared secret $\bar{K}'$ and the received ciphertext ct' is generated (Line 24). However, for an invalid ciphertext, comparison fails with an overwhelming probability, resulting in the generation of a pseudo-random secret K, using a pseudo-random value z and the received ciphertext ct' (Line 22,24). Thus, for invalid ciphertexts, an attacker cannot obtain any information about the decrypted message m', which provides concrete protection against chosen-ciphertext attacks.

2.5 Dilithium

Dilithium is a lattice-based signature scheme secure, whose security is based on the Module LWE (M-LWE) and Module SIS (M-SIS) problem [25]. 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 $q = 2^{23} - 2^{17} - 1$. This choice of parameters allows the use of 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 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.

2.5.1 Algorithmic Description. Refer to Alg.3-4 for a simplified description of 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 Power2Round, HighBits, LowBits, MakeHint and UseHint, whose details can be found in [25]. The key generation procedure simply involves generation of an LWE instance $\mathbf{t}$ (Line 6 in Alg.3). Subsequently, the LWE instance is split into higher and lower order bits $\mathbf{t_1}$ and $\mathbf{t_0}$ respectively (Line 7), 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 the signature and its associated intermediate variables, satisfy a given set of conditions[41]. The message m is first hashed with a public value tr to generate μ (Line 13). The abort loop (Line 21-39) 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 17), while the probabilistic variant randomly samples the seed ρ from a uniform distribution (Line 19). 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 $\mathbf{c} \in R_q$ (Line 25), whose 60 coefficients are either ± 1, while the other 196 coefficients are 0. Subsequently, the challenge $\mathbf{c}$, nonce $\mathbf{y}$ and secret $\mathbf{s}_1$, are used to compute the primary signature component $\mathbf{z}$ (Line 27). Then, a hint vector $\mathbf{h}$ is generated and output as part of the signature σ (Line 33). The abort loop contains several conditional checks (Line 29, 34), 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 5 in Alg.4), which is then compared with the received challenge $\mathbf{c}$, along with other checks (Line 6). If all the checks are satisified, then the verification is successful, else it is a failure. While we have only presented a simplified description of the Dilithium signature scheme, we refer the reader to [25] for a detailed description of the same.

Algorithm 3: Dilithium Signature scheme (Simplified)

procedure KeyGen
  $(seed_A, seed_S, K) \in \mathcal{B} \leftarrow \mathsf{Sample}_U()$;
  $\mathbf{s_1}, \mathbf{s_2} \in (R_q^\ell \times R_q^k) \leftarrow \mathsf{Sample}_B(seed_S)$ ▹ Generate the secrets $\mathbf{s}_1$ and $\mathbf{s}_2$
  $\mathbf{A} \in R_q^{k\times\ell} \leftarrow \mathsf{Expand}(seed_A)$
  $\hat{\mathbf{s_1}} = \mathsf{NTT}(\mathbf{s}_1)$ ▹ Compute NTT of $\mathbf{s}_1$
  $\mathbf{t} = \mathsf{INTT}(\mathbf{A} \circ \hat{\mathbf{s_1}}) + \hat{\mathbf{s_2}}$ ▹ 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
  $\kappa \leftarrow 0$; $(\mathbf{z}, \mathbf{h}) \leftarrow \perp$
  $\hat{\mathbf{s_1}} = \mathsf{NTT}(\mathbf{s}_1), \hat{\mathbf{s_2}} = \mathsf{NTT}(\mathbf{s}_2), \hat{\mathbf{t_0}} = \mathsf{NTT}(\mathbf{t}_0)$
  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
    $\mathbf{y} \leftarrow \mathsf{Sample}_Y(\rho \| \kappa)$
    $\hat{\mathbf{y}} = \mathsf{NTT}(\mathbf{y})$ ▹ $\mathsf{NTT}(y)$
    $\mathbf{w} \leftarrow \mathsf{INTT}(\hat{\mathbf{A}} \circ \hat{\mathbf{y}})$; $\mathbf{w_1} \leftarrow \mathsf{HighBits}(\mathbf{w})$ ▹ $\mathrm{w}_1 = \mathsf{HighBits}(\mathbf{A} \cdot \mathrm{y})$
    $\mathbf{c} \in R_q \leftarrow \mathcal{H}(\mu \| \mathbf{w_1})$ ▹ Generate Sparse Challenge c
    $\hat{\mathbf{c}} = \mathsf{NTT}(c)$ ▹ $\mathsf{NTT}(\mathbf{c})$
    $\mathbf{z} = \mathsf{INTT}(\hat{\mathbf{c}} \circ \hat{\mathbf{s}}_1) + \mathbf{y}$ ▹ $\mathbf{z} = \mathbf{s_1} \cdot \mathbf{c} + \mathbf{y}$
    $\mathbf{r}_0 = \mathsf{LowBits}(\mathbf{w} - \mathbf{c} \cdot \mathbf{s}_2)$
    if $\|\mathbf{z}\|_\infty \geq \gamma_1 - \beta$ or $\|\mathbf{r}_0\|_\infty \geq \gamma_2 - \beta$ **then** ▹ Conditional Checks
      $(\mathbf{z}, \mathbf{h}) = \perp$
      $\kappa = \kappa + 1$
    else
      $\mathbf{h} = \mathsf{MakeHint}(-\mathbf{c} \cdot \mathbf{t}_0, \mathbf{w} - \mathbf{c}\mathbf{s}_2 + \mathbf{c} \cdot \mathbf{t}_0, 2\gamma_2)$
      if $\|\mathbf{c} \cdot \mathbf{t}_0\|_\infty \geq \gamma_2$ or #1's in $\mathbf{h} > \omega$ **then** ▹ Conditional Checks
        $(\mathbf{z}, \mathbf{h}) = \perp$
        $\kappa = \kappa + 1$
      end if
    end if
  end while
  $\sigma = (\mathbf{z}, \mathbf{h}, \mathbf{c})$
end procedure

Algorithm 4: Dilithium Signature scheme (Simplified)

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

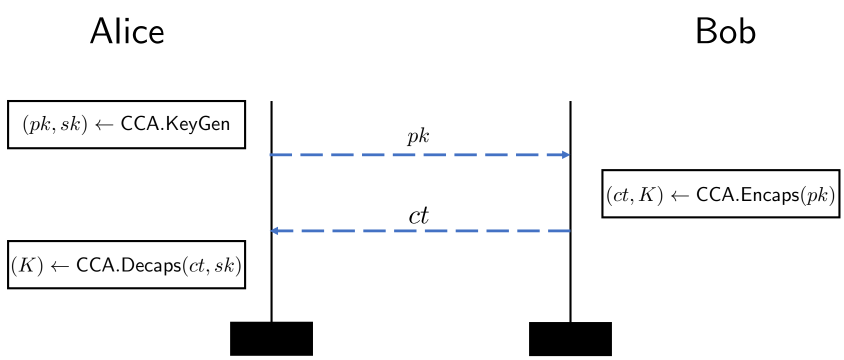

Fig. 2. Key-Exchange protocol using IND-CCA secure Kyber KEM

3 SIDE-CHANNEL ATTACKS ON KYBER KEM

3.1 Nomenclature for Attack Classification

Kyber KEM has been subjected to a variety of side-channel attacks, and the type of attack that can be mounted in a given setting, depends upon several factors such as target procedure, target operation, attack technique, operating mode of Kyber etc. Understanding the applicability of different attacks, requires one to understand the application of Kyber KEM when used for key-exchange (i.e.) within a key exchange protocol.

3.1.1 Application of Kyber KEM for Key-Exchange. Refer to Fig.2 for a 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 using the public key pk to generate the ciphertext ct and the corresponding shared session key K. Bob shares the ciphertext ct with Alice, who then uses her secret key sk to decapsulate the ciphertext (Decaps) to generate the same shared session key K.

This key-exchange protocol can operate in two settings, depending upon the longevity of the key pair (pk, sk) used by Alice.

(1) *Static key setting*: Alice can choose to reuse (pk, sk) for multiple key-exchanges and this is referred to as a static-key setting with ocassional key refreshment (once every X key-exchanges). It is also possible that Alice also uses a single key pair for all key-exchanges without any key refrehsment. For simplicity, we refer to these scenarios together as the static key setting.
(2) *Ephemeral 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 the ephemeral key setting, IND-CCA security is not required, and thus the key-exchange can be carried out only using IND-CPA secure Kyber PKE. Thus, Bob and Alice can 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. However, key exchange in this setting can also be carried out using IND-CCA secure Kyber KEM, and this is left upto the choice of the designer. But, this is generally considered to be overkill, as it is well-known that IND-CPA security is sufficient for ephemeral key-exchange. For instance, the TLS 1.3 protocol mandates ephemeral key-exchange [1], but the post-quantum variant of TLS 1.3 implemented in the Open-Quantum Safe project utilizes IND-CCA secure KEMs for ephemeral key-exchange [17]. Similarly, designers can opt for the stronger IND-CCA secure KEMs, even when performing ephemeral key-exchanges.

The type of attacks on Kyber KEM primarily depends upon the interplay between the following two factors:

(1) Kyber's operating mode - ephemeral/static key setting
(2) Attacker's access to Device Under Test (DUT) - Alice/Bob

3.1.2 Attacking Kyber in Ephemeral Key Setting. In this setting, the secret key sk and sensitive message m are refreshed for every new key exchange. Recovery of m leads to recovery of the session key K. Recovery of sk also leads to recovery of the session key K. This allows the attacker to decrypt future dated secure communication between Alice and Bob, encrypted using the session key K. Thus, in the ephemeral key setting, recovering the message m (message recovery) is equivalent to recovering the secret key sk (key recovery).

(1) *Attacking Alice:* If an attacker has access to Alice's device, then he/she can target the key-generation or decapsulation procedure. Leakage from the key-generation procedure can be exploited to recover the secret key sk. One of the main challenges of targeting the key-generation procedure is that it is probabilistic and generates a new key pair for every execution. Thus, the attacker only has access to a single execution/trace of the key-generation procedure to recover the entire secret key sk. Thus, multi-trace side-channel attacks or fault attacks that require multiple faulty outputs naturally do not apply to the key-generation procedure. However, in the ephemeral key setting, the key generation procedure is executed for every new instance of the key-exchange protocol, thus an attacker has the opportunity to attack key generation, every time a key exchange is initiated by Alice.
An attacker can also exploit leakage from the decapsulation procedure for key recovery (sk) as well as message recovery m. Similar to targeting the key-generation procedure, the attacker only has a single trace/execution of the decapsulation procedure to recover the entire secret key sk. Thus, only single trace/execution attacks apply, while multi-trace attacks do not apply.
(2) *Targeting Bob:* If an attacker has access to Bob's device in the ephemeral key setting, then he/she can target the encapsulation procedure for message recovery (i.e.) m. The encapsulation procedure is also probabilistic, and thus the attacker only has access to a single execution to recover the entire message m.

3.1.3 Attacking Kyber in Static Key Setting. In this setting, recovering the secret key sk is much more attractive for an attacker, since recovering sk results in trivial recovery of the message m for all the key exchanges carried out by Alice using sk. Thus, recovery of a single long-term secret key sk leads to recovery of all the session keys K derived using sk.

(1) *Targeting Alice:* An attacker can target Alice's key-generation and decapsulation procedure. Unlike the ephemeral key setting where the key-generation procedure is executed for every key exchange, key-generation is only executed once every X times, where X is the key refresh rate chosen by the designers based on the application. Thus, the attacker has less opportunity to attack key generation in a static setting with occasional key refreshes, compared to the ephemeral key setting. Moreover, only single trace/execution attacks are applicable to the key-generation procedure.
However, in a static key setting, the decapsulation procedure serves as a better target for the attacker, since it manipulates the same secret key sk for X key exchanges. Thus, an attacker has access to X number of executions of the decapsulation procedure to recover the secret key sk, compared to only a single trace/execution in the ephemeral key setting. Thus, multi-trace attacks apply when targeting the decapsulation procedure in a static key setting.
(2) *Targeting Bob:* If an attacker has access to Bob's device, then he/she can target the encapsulation procedure for message recovery. Similar to the ephemeral key setting, only single trace attacks apply to the encapsulation procedure.

From the aforementioned discussion, we can infer the following:

(1) Key-generation procedure and encapsulation procedure can only be targeted by single trace attacks, irrespective of operating in the static key setting or ephemeral key setting.
(2) In the ephemeral key setting, the decapsulation procedure can be targeted only by single trace attacks. However, in the static key setting, multi-trace attacks apply to the decapsulation procedure.

3.1.4 Attack Scenarios and Characteristics. We therefore discuss side-channel attacks on Kyber KEM based on four scenarios:

(1) Targeting Key-generation Procedure (Single Trace)
(2) Targeting Encapsulation Procedure (Single Trace)
(3) Targeting Decapsulation Procedure in Ephemeral Key Setting (Single Trace)
(4) Targeting Decapsulation Procedure in Static Key Setting (Multi Trace)

For every attack discussed in this work, we also describe its characteristics based on the following parameters:

(1) *Attacker's ability to communicate with DUT* (DUT_IO_Access): In this respect, we identify two categories:
    (a) Observe_DUT_IO: We assume an attacker who can only passively observe the DUT's communication channel (I/O), but cannot actively communicate with the DUT. In this scenario, the attackers can only observe/affect the DUT's behaviour for valid key exchanges with other parties. The attacker cannot trigger the operation of the DUT.
    (b) Communicate_DUT_IO: We assume an attacker who can passively observe the DUT's communication channel, while also being able to actively communicate with the DUT. For instance, when targeting Alice, an attacker can attempt to establish a key exchange, and submit ciphertext queries to observe behaviour of Alice. Similarly, when targeting Bob, an attacker can attempt to perform a valid key exchange with Bob, to observe the behavior of his DUT. This ability to communicate with the DUT provides the attacker

the opportunity to profile the side-channel behaviour of the DUT when processing the attacker's chosen inputs. As we show later in Section 3.6, this serves as an advantage for an attacker, leading to more variety of attacks.

(2) *Profiling and Access to clone device* (Profile_*Requirement*): In this respect, we can categorize attacks into three categories:
   (a) Profiled_With_Clone: This category includes profiled attacks, which work with side-channel templates built using leakage from a clone device of the DUT. Thus, the attacker requires access to a clone device, which he/she can fully control, including the secret key. The attacker can construct elaborate side-channel templates, using leakage from the clone device, for every operation done on the DUT.
   (b) Profiled_Without_Clone: This category includes profiled attacks, which can work with templates constructed using leakage, directly from the DUT. Thus, these attacks do not require access to a clone device.
   (c) Non_Profiled: This category includes attacks that do not utilize any side-channel templates, thus these attacks also do not require access to a clone device.

(3) *Number of traces* (No_Traces): This characteristic denotes the number of traces from the DUT, required to perform message recovery/key recovery. We do not take into account the number of traces, required to construct side-channel templates. We only consider the number of executions of the DUT to carry out the attack. The exact number of traces for key/message recovery depends upon the experimental setup, and therefore we only provide approximate numbers for the same in this paper, while the main emphasis is on the scale of the number of traces, rather than the exact number.

(4) *Signal to Noise Ratio* (SNR): This characteristic indicates the robustness of the side-channel attack to measurement noise in the acquired traces. We identify two categories: Low_SNR and High_SNR. We clarify that the SNR comparison is only qualitative and that attacks that work over multiple traces typically require lower SNR, compared to attacks that work with single traces.

We therefore define the characteristic of each side-channel attack on Kyber using the following tuple: (DUT_IO_Access, Profile_*Requirement*, No_Traces, SNR). For example, a tuple (Observe_DUT_IO, Profiled_With_Clone, 1, High_SNR) indicates a side-channel attack with the following characteristics: one that does not require to communicate with the DUT, requires a clone device to construct templates, only requires a single trace from the DUT, and requires a reasonably high SNR in the collected measurements. We believe this representation captures the high-level features about the attack, and allows us to categorize existing attacks into different categories.

To explain the different attacks, we utilize the algorithm of IND-CPA secure Kyber PKE in Alg.1 and the algorithm of IND-CCA secure Kyber KEM in Alg.2. In this paper, we consider power/EM side-channels to be equivalent, as attacks exploiting the power (resp. EM) side-channel can be easily adapted to exploit the EM (power) side-channel, albeit with a difference in the success rate and effort to carry out the attack.

3.2 SCA on Key Generation

We now discuss single trace attacks applicable to the key-generation procedure of Kyber KEM.

3.2.1 Soft-Analytical Side-Channel Analysis (SASCA). In a single power/EM trace, the attacker has to extract as much information as possible from a single trace, to recover the target variable. In this respect, Soft-Analytical Side-Channel Analysis (SASCA) acts as a very potent tool to perform single-trace attacks. They are profiled attacks, which work by templating leakage from multiple sequential operations, directly processing the secret variable. Subsequently, to carry out the attack, the attacker obtains a single trace, which is then matched with all the templates, and the template matching information is combined to recover the secret key. The ability of SASCA for key recovery was initially studied for symmetric key cryptographic schemes such as AES [73], however, its applicability to lattice-based schemes has also been studied by several works [39, 53, 55]. We can classify existing attacks based on SASCA into two categories, based on the target operation.

Targeting NTT: Primas *et al.* [55] showed that a single power/EM trace from the NTT operation operating over a secret variable $\mathbf{x}$, can be used to recover $\mathbf{x}$ (i.e.) input to the NTT. A close observation of the algorithm of IND-CPA Kyber PKE (Alg.1) reveals that NTT is computed over several sensitive intermediate variables. In particular, within the key-generation procedure, NTT is computed over the secret key $\mathbf{s} \in R_q^k$ (Line 7 of KeyGen). Thus, leakage from this NTT operation can be exploited by SASCA to recover its input (i.e.) $\mathbf{s}$.

The attack works in two phases - (1) Profiling Phase and (2) Key Recovery Phase.

(1) *Profiling Phase*: Side-channel templates are constructed using leakage from the clone device (Profiled_With_Clone), for several intermediate computations within the NTT. Some of these operations include storing/loading of input and output of butterfly operation, modular addition, modular subtraction, and modular multiplication.
(2) *Key Recovery Phase*: Once templates are constructed, the attacker obtains a single trace corresponding to leakage from the target NTT operation. The trace is segmented based on the targeted internal operations, which are then matched with the appropriate templates. Subsequently, results from the template matching are modeled into a factor graph based on the NTT implementation. The factor graph is fed into the Belief Propagation algorithm [52] which combines leakage from all the intermediate variables, to recover the secret key $\mathbf{s}$.

The first SASCA-based single trace attack on lattice-based schemes was proposed by Primas *et al.* [55] on a generic Ring-LWE-based PKE scheme, implemented on the ARM Cortex-M4 microcontroller. The proposed attack required over a million templates for successful key recovery. However, the subsequent work of Pessl and Primas [53] proposed optimizations to reduce the number of templates to just a few hundred (≈ 200), especially when the coefficients of the input to the target NTT belong to a small range. This is precisely the case with the NTT over the secret $\mathbf{s} \in R_q^k$ in the KeyGen procedure (Line 7 in Alg.1). We refer to these attacks targeting NTT using SASCA by the label SASCA_NTT. Their characteristic can be defined by the following tuple: (Observe_DUT_IO, Profiled_With_Clone, 1, High_SNR).

Very recently, Li *et al.* [39] proposed single trace attacks on the reference implementation of the Toom-Cook polynomial multiplier, used in Saber KEM. The aforementioned attacks clearly demonstrate the ability of SASCA-style attacks to break different algorithms for polynomial multiplication, when used in lattice-based schemes.

Countermeasure: We refer to the work of Ravi *et al.* [63] who proposed generic shuffling and masking countermeasures with varying granularity, to protect the NTT against single trace attacks. They proposed a range of shuffling countermeasures that provide a well-defined trade-off between shuffling entropy (security) and performance. They also proposed masked NTTs, which randomize the twiddle factors used in the NTT operation. This has the effect of randomizing computations within the NTT, which deters the success rate of SASCA-type attacks. For more details, we refer to [63] for the proposed masking and shuffling countermeasure for the NTT. Recently, the security offered by the shuffled NTT countermeasures was studied in a detailed manner by Hermelink *et al.* [35]. We refer to these countermeasures for the NTT together using the label Shuffled_Masked_NTT.

Targeting KECCAK: KECCAK is used as a building block in several lattice-based schemes including Kyber KEM. In particular, it is used as a pseudo-random function (PRF) and a pseudo-random number generator (PRNG) across all the three procedures of Kyber KEM (KeyGen, Encaps and Decaps). In the KeyGen procedure, KECCAK is used as a PRNG to expand a small secret seed $seed_B$ into a string of pseudo-random bits (Line 5 in Alg.1), which is subsequently used to sample the secret $\mathbf{s}$. Given the sequential nature of the KECCAK operation, it serves as an ideal target for SASCA.

In this respect, Kannwischer *et al.* [37] demonstrated single trace SASCA on KECCAK instances, which can be used to recover their inputs. Thus, targeting the KECCAK operation over the secret $seed_B$ (Line 5 in Alg.1), can be used to recover $seed_B$, whose knowledge can be used to recover the secret $\mathbf{s}$. Though the attack was only demonstrated over simulated traces, the attack in principle is applicable to software implementations, particularly on embedded microcontrollers such as the ARM Cortex-M4. We henceforth refer to the attack using the label SASCA_KECCAK. The templates required to perform the attack can only be built using leakage from a clone device. The attack characteristic can be defined by the tuple: (Observe_DUT_IO, Profiled_With_Clone, 1, High_SNR). KECCAK is extensively utilized as a PRF and PRNG in several lattice-based KEMs and also extensively within hash-based signatures, thus the SASCA_KECCAK attack is also applicable to other PQC schemes, as discussed in [37].

Countermeasure: Similar to the NTT operation, KECCAK can be protected from SASCA_KECCAK style attacks through shuffling. However, we are not aware of prior work that investigates the cost and effectiveness of partial or full shuffling of KECCAK instances in lattice-based schemes. We refer to the shuffling countermeasure for KECCAK as Shuffled_KECCAK throughout this paper.

Though SASCA-based attacks targeting the NTT and KECCAK instances can work with single traces, they suffer from a few downsides:

(1) *Requirement of Elaborate Profiling*: Several hundred precisely built templates for low-level arithmetic operations are required for a successful attack. This requires the attacker to have detailed information about the target and its internal operations.
(2) *Requirement of high Signal to Noise Ratio (SNR)*: The attack typically requires a relatively high SNR for full key recovery, which is typical of single trace attacks. Thus, incorporation of low-cost countermeasures such as jitter could already be sufficient to significantly deter the success rate of the attack.
(3) *Applicability of attack to noisy devices*: The aforementioned attacks have only been demonstrated on embedded microcontrollers such as the ARM Cortex-M4 with high SNR. But their applicability to more advanced processors with inherently low SNR is not clear. Moreover, hardware implementations with inherent parallelism introduce significant algorithmic noise, which can also significantly deter attack success rate.

Refer to Tab.1 for a tabulation of all side-channel attacks on the key-generation procedure of Kyber KEM.

3.3 SCA on Encapsulation

Similar to the key-generation procedure, the encapsulation procedure is also probabilistic and is therefore only susceptible to single trace message recovery attacks, in both the ephemeral key setting as well as static key setting. We now discuss single trace attacks applicable to the encapsulation procedure of Kyber KEM.

3.3.1 SASCA. The encapsulation procedure is also susceptible to SASCA-based attacks.

Targeting NTT (SASCA_NTT): Leakage from the NTT over the ephemeral secret $\mathbf{r} \in R_q^k$ (Line 17 of Encrypt in Alg.1) can be exploited to recover $\mathbf{r}$ in a single trace. Recovery of $\mathbf{r}$ leads to straightforward recovery of the message m for a valid ciphertext $ct = (\mathbf{u}, \mathbf{v})$ in the following manner:

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

The attack can only be carried out using templates built from the clone device since the attacker does not have knowledge of the internal variables of the target computation (i.e.) $\mathsf{NTT}(\mathbf{r})$. Thus, the attack characteristic can be defined by the following tuple: (Observe_DUT_IO, Profiled_With_Clone, 1, High_SNR).

Targeting KECCAK (SASCA_KECCAK): KECCAK is used as a PRF as well as a PRNG within the encapsulation procedure. In the Encaps procedure, it is used as a PRF (denoted as $\mathcal{G}$) to generate the pre-key $\bar{K}'$ and seed r, using the sensitive message m and hash of the public key pk (Line 10 in Alg : $\mathsf{CCA_transform}$). Thus, exploiting leakage from this KECCAK instance leads to recovery of the message m in a single trace.

In the Encrypt procedure, the KECCAK operation is used as a PRNG, to expand a small secret seed (i.e.) $seed_R$ (32 bytes) into a string of pseudorandom bits, which are further used to sample the ephemeral secrets $\mathbf{r}$, $\mathbf{e}_1$ and $\mathbf{e}_2$ (Lines 14-16 in Alg.1). Exploitation of leakage from this KECCAK instance leads to recovery of $seed_R$ in a single trace, whose knowledge can be used to recover $\mathbf{r}$ and therefore the message m. Similar to SASCA on the NTT, the attack can be defined using the tuple: (Observe_DUT_IO, Profiled_With_Clone, 1, High_SNR).

Countermeasure: Shuffling and masking the NTT (Shuffled_Masked_NTT) as well as shuffling the KECCAK operation (Shuffled_KECCAK) provides concrete protection against attacks relying on SASCA.

3.3.2 Targeting Message Encoding. KEMs based on the LWE/LWR-based problem such as Kyber, inherently involve bitwise manipulation of the message m. During encryption, the message $m \in \mathcal{B}^{32}$ is encoded one bit at a time into a polynomial $\mathbf{m} \in R_q$ within the Encode procedure (Line 20 of Encrypt in Alg.1). This behaviour has been exploited by several power/EM side-channel attacks for message recovery [4, 69, 77].

The first such attack was demonstrated by Amiet *et al.* [4] targeting the message encoding operation in NewHope KEM, a Ring-LWE based KEM. The encoded message polynomial has only two possible coefficients (i.e.) $\mathbf{m}[i] = \lceil q/2 \rfloor$ for $m_i = 1$ and $\mathbf{m}[i] = 0$ for $m_i = 0$. For a non-zero modulus q, the difference in Hamming Weight of $\mathbf{m}[i]$ when $m_i = 0/1$ (i.e.) $\mathbf{m}[i] = 0$ or $\mathbf{m}[i] = q/2$, can be easily distinguished through the power/EM side-channel. Thus, leakage from manipulation of these encoded coefficients (Line 20) can be used to recover the message one bit at a time, from a single trace.

The attack works in two phases - (1) Profiling Phase and (2) Key Recovery Phase.

(1) *Profiling Phase*: The attacker builds templates for all message bits $m_i = 0$ and $m_i = 1$ for $i \in [0, 256)$. The templates can be constructed in two ways. If the attacker can communicate with the DUT (Communicate_DUT_IO), then he/she can perform several valid key-exchanges with the DUT to build side-channel templates for all message bits. Thus, the templates can be built directly on the DUT (Profiled_Without_Clone). However, if the attacker cannot communicate with the DUT (Observe_DUT_IO), then templates have to be built on a clone device to carry out the attack (Profiled_With_Clone).

(2) *Key Recovery Phase*: The attacker obtains a single attack trace, and divides it into smaller segments, corresponding to the individual message bits m_i for $i \in [0, 256)$. These segments are matched with the corresponding templates $m_i = 0$ and $m_i = 1$, to recover the entire message in a single trace.

While the attack was originally demonstrated on NewHope KEM, subsequent works [69, 77] have generalized the same attack to multiple lattice-based KEMs including Kyber KEM. We refer to these attacks using the label Message_Encode.

Since it is a single-trace attack, masking does not serve as a concrete countermeasure against the attack. In fact, attacks have been demonstrated exploiting similar bitwise manipulation of the message, on implementations protected with first-order and higher-order masking countermeasures [47, 49]. These attacks show that an attacker can exploit leakage from all the individual shares of the message bits for single-trace message recovery. The attack in effect does not really break the security guarantees of the masked implementation but is merely a second-order attack on a first-order masked implementation. However, this attack is of interest, because it is expected that the number of traces required for the attack increases exponentially with the masking order. However, Ngo *et al.* [47] showed that single trace message recovery is also possible on a first-order masked implementation, similar to the unprotected implementation. Similarly, the same attack was extended to higher orders by the same authors [49], thereby clearly demonstrating that masking alone, does not deter message recovery when the message is manipulated in a bitwise fashion. Though these attacks exploited the message decoding operation within the decoding procedure of Saber KEM (equivalent to Line 28 of Decrypt in Alg.1), the same attacks also apply to the encoding procedure.

We refer to these attacks applicable to the masked message encoding procedure as Masked_Message_Encode. All the aforementioned attacks have been demonstrated on the ARM Cortex-M4 microcontroller, where the encoding operation is done in a sequential manner, one bit at a time. However, the applicability of these single-trace attacks to parallelized implementations is not clear. Moreover, leakage from the manipulation of single message bits only spans for a single clock cycle. Thus, exploitation of such fine-grained leakage requires traces with sufficiently high SNR for message recovery.

The characteristic of both the Message_Encode and Masked_Message_Encode attacks the encapsulation procedure can be described using these two tuples: (Observe_DUT_IO, Profiled_With_Clone, 1, High_SNR) and (Communicate_DUT_IO, Profiled_Without_Clone, 1, High_SNR).

Countermeasure: Single trace attacks on the message encoding operation can also be concretely prevented by shuffling the message encoding operation, as proposed by [4]. Shuffling ensures that the attacker can recover all the bits of the message, but not the order of message bits. This concretely prevents message recovery, since full shuffling has an entropy of (n!) for an n-bit message (i.e.) 256! for $n = 256$ bits, which is beyond brute force for an attacker. We henceforth refer to this countermeasure using the label Shuffled_Encode.

Refer to Tab.1 for a tabulation of all side-channel attacks on the encapsulation procedure of Kyber KEM.

3.4 SCA on Decapsulation Procedure in Ephemeral Key Setting

In this section, we discuss single trace side-channel attacks applicable to the decapsulation procedure in the ephemeral key setting. We recall that both message recovery and key recovery attacks are possible, and that message recovery has the same impact as that of performing key recovery in the ephemeral key setting. We reiterate that IND-CPA secure KEMs are sufficient for concrete security in ephemeral key exchanges, but we consider usage of IND-CCA secure KEMs in the ephemeral key setting, as the operating mode of KEM is purely the designer's choice [17]. When IND-CPA secure PKE of Kyber is used for ephemeral key-exchange, then the attacker can only target leakage from the decryption procedure for key recovery/message recovery (Line 17 of Decaps in Alg.2). However, when IND-CCA secure KEM is used, the attacker can target leakage from any operation after the decryption procedure for key recovery/message recovery (Lines 18-24 of Decaps in Alg.2).

3.4.1 *SASCA.*

The encapsulation procedure is also susceptible to SASCA-based attacks.

Targeting NTT in the Decryption Procedure (SASCA_NTT): An attacker can target the INTT operation over $\hat{g}'$ (i.e.) the product of NTT of ciphertext component $\mathbf{u}'$ and the NTT of secret $\mathbf{s}$ (Line 27 of Decrypt in Alg.1). Recovery of $\hat{g}'$ results in trivial recovery of the secret key $\mathbf{s}$, since the ciphertext component $\mathbf{u}$ is known to the attacker. The attack can be carried out using templates from a clone device. We thus define the characteristic of the attack using the following tuple: (Observe_DUT_IO, Profiled_With_Clone, 1, High_SNR).

Targeting NTT in the Re-encryption Procedure (SASCA_NTT): Apart from targeting NTT in the decryption procedure, an attacker can also target NTT instances in the re-encryption procedure, in the same manner as targeting NTT instances in the encapsulation procedure. In this respect, an attacker can target the NTT operation over the ephemeral secret $\mathbf{r}$ within the re-encryption procedure (Line 20 in Alg.2). This enables the attacker to recover $\mathbf{r}$, whose knowledge can be used to recover the message m for a given target ciphertext ct.

There is however one subtle difference with respect to profiling when compared to the same attack on the encapsulation procedure. If the attacker is able to communicate with the decapsulation procedure (Communicate_DUT_IO), he/she can build templates using leakage from decapsulation of valid ciphertexts, directly from the DUT.

This can be done in the following manner: The attacker can construct valid ciphertexts ct_i for $i \in [0, T-1]$ for which the attacker knows the value of the ephemeral secret $\mathbf{r}_i$ for $i \in [0, T-1]$. Leakage from the decapsulation of these ciphertexts can be used to build templates for all internal operations within the NTT over $\mathbf{r}$. In this scenario, we define the attack characteristic using the tuple: (Communicate_DUT_IO, Profiled_Without_Clone, 1, High_SNR). However, if the attacker cannot communicate with the DUT, then he/she requires access to a clone device for profiling. In this scenario, we define the attack characteristic using the tuple: (Observe_DUT_IO, Profiled_With_Clone, 1, High_SNR).

Targeting KECCAK after Decryption Procedure (SASCA_KECCAK): An attacker can target KECCAK instances after the decryption procedure, similar to the attack on KECCAK instances in the encapsulation procedure. One can target the KECCAK instance used as a PRF to generate the pre-key $\bar{K}'$ (Line 18 of Decaps in Alg.2) for message recovery. Similarly, the attacker can target the KECCAK used as a PRNG in the re-encryption procedure (Line 14 of Encrypt in Alg.1) to recover $\mathbf{r}$, leading to message recovery. It is important to note that both operations primarily depend upon the message m'. Thus, an attacker who can communicate with the DUT can control m' during valid key exchanges and build templates directly on the DUT. In this scenario, we define the attack characteristic using the tuple: (Communicate_DUT_IO, Profiled_Without_Clone, 1, High_SNR). However, if the attacker cannot communicate with the DUT, then templates can only be built on the clone device (i.e.) (Observe_DUT_IO, Profiled_With_Clone, 1, High_SNR).

Countermeasure: Shuffled_Masked_NTT as well as Shuffled_KECCAK countermeasures can be used to protect against SASCA style attacks.

3.4.2 Targeting Message Decoding. Similar to the message encoding operation within the Encrypt procedure, the message decoding operation within the decryption procedure (Decrypt) also performs bitwise manipulation of the decrypted message m' (Line 28 of Decrypt in Alg.1). The erroneous message polynomial $\mathbf{m'} \in R_q$ is decoded into the message $m' \in \mathcal{B}^{32}$, one coefficient at a time. This bitwise manipulation was shown to be exploitable by Ravi *et al.* [58] using single trace attacks on Kyber KEM. We refer to this attack using the label Message_Decode. Subsequently, Ngo *et al.* [47] demonstrated a similar message recovery attack on the masked decoding procedure in Saber KEM, which has also been extended to higher order masked implementations as well [49, 58]. We refer to this attack using the label Masked_Message_Decode.

If the attacker can communicate with the DUT (Communicate_DUT_IO), then templates for the message can be built directly on the DUT. Thus, the attack characteristic in this scenario is (Communicate_DUT_IO, Profiled_Without_Clone, 1, High_SNR). If the attacker cannot communicate with the DUT (Observe_DUT_IO), then templates have to be built on a clone device. Thus, the attack characteristic in this scenario is (Observe_DUT_IO, Profiled_With_Clone, 1, High_SNR).

Similar to the Message_Encode attacks on the message encoding procedure, an attack on the decoding procedure also requires less noise in the measurements. Moreover, leakage from manipulation of single message bits only spans for a single clock cycle. Thus, adapting the attack to advanced platforms with inherent measurement or algorithmic noise is not very trivial.

Countermeasure: Similar to shuffling the message encoding procedure, shuffling the message decoding procedure provides concrete protection against such single-trace message recovery attacks, as proposed in [4]. We henceforth refer to this countermeasure as Shuffled_Decode.

Apart from the aforementioned attacks, we observe that single attacks on the encapsulation procedure also apply to the decapsulation procedure. This is because all operations targeted by side-channel attacks on the encapsulation procedure are also performed in the decapsulation procedure, due to the use of the FO transform (Refer Sec.3.3).

Refer to Tab.1 for a tabulation of all side-channel attacks on the decapsulation procedure in the ephemeral key setting of Kyber KEM.

3.5 SCA on Decapsulation Procedure in Static Key Setting

In the static key setting, the decapsulation procedure manipulates the same secret key *sk* for multiple key exchanges. Thus, the attacker has access to multiple traces from the decapsulation procedure to perform key recovery and message recovery. Clearly, single trace attacks applicable to the decapsulation procedure in the ephemeral key setting, are also applicable in the static key setting (Refer Sec.3.4). Thus, we only discuss those attacks that utilize multiple traces from the decapsulation procedure for key recovery.

3.5.1 Correlation Power Analysis (CPA). We first discuss attacks that assume an Observe_DUT_IO attacker, who can only passively monitor the I/O of the DUT performing decapsulation. In this respect, Mujdei *et al.* [45] performed an extensive study on CPA style attacks and their applicability to different polynomial multiplication strategies, including NTT. Kyber adopts an *incomplete* NTT for polynomial multiplication, for efficiency reasons (i.e.) it only computes $(log_2(n) - 1)$ layers of the NTT for an $n - 1$ degree polynomial. Thus, the output of the incomplete NTT is nothing but a sequence of linear polynomials with degree 1. The decryption procedure computes the incomplete NTT of $\mathbf{u'} \in R_q^k$ (Line 25 in Alg.1), which is followed by a pointwise multiplication with the coefficients of the NTT transformed secret

$\hat{\mathbf{s}} \in R_q^k$ (i.e.) $(\hat{\mathbf{u}}' \circ \hat{\mathbf{s}})$ in Line 26 of Alg.1. This pointwise multiplication is performed using several 2-coefficient schoolbook multiplication operations.

Mujdei *et al* [45] showed that leakage from the schoolbook polynomial multiplications after the incomplete NTT can be exploited through conventional CPA style attacks. The presented attack is a non-profiled attack, similar to other CPA style attacks, and requires ≈ 200 power traces to recover all the coefficients of $\hat{\mathbf{s}}$, which enables full key recovery. Similarly, Chen *et al.* [16] demonstrated a non-profiled CPA attack targeting the school-book polynomial multiplication over NTT transformed polynomials used in the signing procedure of Dilithium for key recovery. Their attack can also be adapted to Kyber, which requires less than 200 power traces for key recovery. We refer to these attacks using the label NTT_Leakage_CPA. Since these attack work over multiple traces, they can still work with low SNR. The attack characteristic can be described using the following tuple: (Observe_DUT_IO, Non_Profiled, ≈ 200, Low_SNR).

Countermeasure: Similar to typical CPA style attacks, masking serves as a concrete countermeasure, which splits the sensitive variable into multiple shares, and operates over each of them independently throughout the implementation. We refer to this countermeasure using the label Masking [13, 33]. However, one of the disadvantages of masking is its high cost ($\approx 2.5 - 3\times$) as clearly shown in [13, 33].

In the following, we discuss those attacks which assume a Communicate_DUT_IO attacker, who can submit ciphertexts of his/her choice for decapsulation by the DUT. Several works have shown that such an attacker can exploit leakage from the decapsulation procedure in different ways to carry out key recovery attacks [58, 65, 77]. This forms the largest category of attacks applicable to lattice-based KEMs such as Kyber KEM, which we refer to as side-channel assisted chosen-ciphertext attacks (SCA-assisted CCA).

3.6 Side-Channel Assisted Chosen-Ciphertext Attacks

Kyber KEM is IND-CCA secure, and therefore enjoys concrete theoretical security guarantees against classical chosen-ciphertext attacks, which query the target with malformed/handcrafted chosen ciphertexts. This is primarily due to the attacker's inability to access any information about sensitive intermediate variables in the decapsulation procedure. However, an attacker who can utilize side-channel leakage can realize a practical *oracle*, to obtain critical information about secret-dependent internal variables within the decapsulation procedure for chosen-ciphertexts, leading to key recovery.

In the following, we discuss the different types of SCA-assisted CCAs applicable to Kyber KEM. Their modus operandi is given as follows: The attacker queries the decapsulation procedure with handcrafted ciphertexts. These ciphertexts are crafted such that the decrypted message m' is very closely related to a targeted portion of the secret key, or in a few cases, the entire secret key. The attacker utilizes leakage from the decapsulation procedure to recover information about m', thereby realizing a practical side-channel *oracle*. Such information obtained over several carefully crafted ciphertexts enables full key recovery. Following are the major sub-categories of side-channel oracle-based CCAs.

(1) Binary Plaintext-Checking (PC) Oracle-Based SCA
(2) Parallel Plaintext-Checking (PC) Oracle-Based SCA
(3) Decryption-Failure (DF) Oracle-Based SCA
(4) Full-Decryption (FD) Oracle-Based SCA

3.6.1 Binary Plaintext-Checking (PC) Oracle-Based SCA. An attacker constructs ciphertexts, so as to ensure that m' (Line 28 in Alg.1) only depends upon a single targeted coefficient of the secret key. Side-channel leakage from the subsequent operations processing m' (Lines 18-20 in Alg.2) are used to instantiate a *Plaintext-Checking* (PC) oracle for key recovery. We briefly explain the PC oracle-based SCA on Kyber KEM, and the same attack can also be adapted to other LWE/LWR-based schemes such as Saber, as shown in [65]. Referring to Alg.1, the attacker chooses a very sparse ciphertext $ct = (\mathbf{u}, \mathbf{v}) \in (R_q^k \times R_q)$ as follows:

$$\mathbf{u}_i = \begin{cases} U \cdot x^0 & \text{if } i = 0, \\ 0 & \text{if } 1 \leq i \leq k-1 \end{cases} \quad (4)$$

$$\mathbf{v} = V \cdot x^0 \quad (5)$$

where $(U, V) \in \mathbb{Z}^+$. For this chosen-ciphertext, each bit of the decrypted message m' (i.e.) m'_i for $i \in [0, n-1]$ is given as:

$$m'_i = \begin{cases} \text{Decode}(V - U \cdot \mathbf{s}_0[0]), & \text{if } i = 0 \\ \text{Decode}(-U \cdot \mathbf{s}_0[i]), & \text{for } 1 \leq i \leq n-1 \end{cases} \quad (6)$$

Thus, every bit m'_i is only dependent on a single corresponding secret coefficient of $\mathbf{s}_0$ (i.e.) $\mathbf{s}_0[i]$. The attacker can chooses tuples (U, V) such that:

$$m'_i = \begin{cases} \mathcal{F}(\mathbf{s}_0[0]), & \text{if } i = 0 \\ 0, & \text{for } 1 \leq i \leq n-1 \end{cases} \quad (7)$$

Now, m' can only take two possible values (i.e.) $m' = 0/1$, whose value depends upon a single secret coefficient $\mathbf{s}_0[0]$. Thus, $m' = 0/1$ for different tuples (U, V) can be used as a binary distinguisher for every possible candidate of $\mathbf{s}_0[0]$. Recovery of $m' = 0/1$ is done through side-channels (side-channel based PC oracle). In a similar manner, attackers can build ciphertexts to recover the other secret coefficients, one at a time, leading to full key recovery.

In this respect, D'Anvers *et al.* [22] presented the first SCA-assisted CCA, targeting a non-constant time implementation of LAC, a Ring-LWE based KEM [40]. The targeted design utilized a non-constant time implementation of the BCH decoding procedure. D'Anvers *et al.* [22] showed that the time taken to decode $m' = 0$ after decryption, is much smaller than the time taken to decode $m' = 1$. This timing side-channel information was used to recover m' for chosen-ciphertexts, which resulted in key recovery in a few thousand chosen-ciphertexts for LAC KEM.

Subsequently, Ravi *et al.* [65] generalized the attack to several constant-time implementations of LWE/LWR-based KEMs, including Kyber KEM. utilizing the power/EM side-channel. They observed that a single-bit difference in m' $(0/1)$, uniformly randomizes all subsequent operations after decryption (i.e.) due to the use of hash functions in the decapsulation procedure (Lines 18-20 in Alg.2). Thus, power/EM side-channel leakage from any of these operations can be used to realize a practical binary PC oracle to distinguish between $m' = 0$ and $m' = 1$.

The attack works in two phases - (1) Pre-processing Phase and (2) Key Recovery Phase.

(1) *Pre-processing Phase*: In this phase, side-channel templates are constructed for leakage from the re-encryption procedure for $m' = 0$ and $m' = 1$, using a simple Welch's t-test. Templates can be built directly on the DUT, by querying with valid ciphertexts corresponding to $m' = 0$ and $m' = 1$. Since leakage from the re-encryption procedure depends upon both m' and pk, templates have to be built for every new key pair (pk, sk).
(2) *Key Recovery Phase*: In this phase, the attacker obtains single traces corresponding to all the chosen-ciphertexts (Eqn.5) and subsequently, each attack trace is classified as either $m' = 0/1$ through simple template matching. This information is sufficient for full key recovery.

More recently, Ueno *et al.* [72] studied the applicability of the aforementioned attack to all KEMs in the NIST standardization process, and demonstrated that almost all KEMs were susceptible to similar binary PC oracle based chosen-ciphertext attacks.

One of the main advantages of the attack is that it can be carried out without any knowledge or very minimal knowledge about the implementation. Moreover, any operation after the decryption procedure (Lines 18-20 of Decaps in Alg.2) can be exploited to instantiate a practical PC oracle for key recovery, which amounts to a few hundred to few thousand leakage points. Thus, the attack can also work with low SNR, due to a large number of leakage points, available for exploitation. However, the attack only recovers a single bit of information about the secret key in each query. Thus, full key recovery requires a few thousand ($\approx 1k - 3k$) chosen-ciphertext queries for Kyber KEM. More recently, few works have proposed improved methods to construct chosen-ciphertexts to reduce the number of queries for key recovery [9, 56], and also to perform efficient key recovery in the presence of a non-perfect side-channel binary PC oracle [68].

We therefore refer to the aforementioned attacks together using the label Binary_PC_Oracle_CCA attack. We define the attack characteristic using the following tuple: (Communicate_DUT_IO, Profiled_Without_Clone, $\approx 1k - 3k$, Low_SNR).

Countermeasure: Masking the decapsulation procedure serves as a concrete countermeasure against the Binary_PC_Oracle_CCA attack (Masking [13, 33]). While higher-order attacks are still possible, they incur a corresponding exponential increase in the number of traces for key recovery.

3.6.2 *Parallel Plaintext-Checking (PC) Oracle-Based SCA.*

Very recently, Rajendran *et al.* [57] and Tanaka *et al.* [71] demonstrated improved PC oracle based side-channel attacks, which are capable of more than one bit of information per query. They demonstrated the ability to recover a generic P number of bits of information about the secret key in a single query ($P \in \mathbb{Z}^+$) through the construction of modified ciphertexts $ct = (\mathbf{u}, \mathbf{v}) \in (R_q^k \times R_q)$ as follows:

$$\mathbf{u}_i = \begin{cases} U \cdot x^0 & \text{if } i = 0, \\ 0 & \text{if } 1 \le i \le k-1 \end{cases} \tag{8}$$

$$\mathbf{v} = V \cdot \Big(\sum_{i=0}^{i=(P-1)} x^i\Big) \tag{9}$$

where $(U, V) \in \mathbb{Z}^+$. For this chosen-ciphertext, each bit of the decrypted message m' (i.e.) m'_i for $i \in [0, n-1]$ is given as:

$$m'_i = \begin{cases} \text{Decode}(V - U \cdot \mathbf{s}_0[i]), & \text{for } i \in [0, P-1] \\ \text{Decode}(-U \cdot \mathbf{s}_0[i]), & \text{for } P \le i \le n-1 \end{cases} \tag{10}$$

Thus, every bit m'_i for $i \in [0, P-1]$ is only dependent on a single corresponding secret coefficient of $\mathbf{s}_0$ (i.e.) $\mathbf{s}_0[i]$. The attacker can chooses tuples (U, V) such that:

$$m'_i = \begin{cases} \mathcal{F}(\mathbf{s}_0[i]), & \text{for } i \in [0, P-1] \\ 0, & \text{for } P \le i \le n-1 \end{cases} \tag{11}$$

Thus, the first P bits of m' (i.e.) m'_i for $i \in [0, P-1]$ are now dependent on the corresponding coefficients of $\mathbf{s}_0$ (i.e.) $\mathbf{s}_0[i]$ for $i \in [0, P-1]$, while all the other bits are fixed to 0. Thus, each of the P message bits serves as a binary distinguisher for the corresponding coefficient of $\mathbf{s}_0$. An attacker who can recover these P message bits per query can realize a P-way parallel PC oracle for key recovery. We refer to it as the Parallel_PC_Oracle_CCA attack.

The realization of such a P-way parallel PC oracle, reduces the number of attack traces/queries for key recovery, by a factor of P, compared to the Binary_PC_Oracle_CCA attack [22, 65]. In this respect, Rajendran *et al.* [57] and Tanaka *et al.* [71] experimentally demonstrate that there is enough information present in power/EM side-channel leakage from the re-encryption procedure to distinguish between 2^P possible values of the message m' in a single trace for $P < 10$. For $P = 10$, full key recovery can be done in ≈ 200 traces. However, higher values for P, if achievable, can further reduce the number of attack traces for key recovery.

However, it is important to note that increasing P, also exponentially increases the number of templates to be built in the pre-processing phase (2^P), while the number of traces in the attack phase only reduces linearly by a factor of P. If an attacker has access to a clone device (Profiled_With_Clone), then the template phase can be completely taken offline, allowing to arbitrarily increase P to reduce the number of queries to the DUT. However, if there is no clone device access, then the attacker has to identify a trade-off between traces for the pre-processing phase, and the key recovery phase. We therefore define the characteristic of Parallel_PC_Oracle_CCA attack using the following tuples: (Communicate_DUT_IO, Profiled_Without_Clone, $\approx 300-500$, Low_SNR), (Communicate_DUT_IO, Profiled_With_Clone, $\approx 100-200$, Low_SNR).

Countermeasure: Similar to the Binary_PC_Oracle_CCA attack, masking the entire decapsulation procedure serves as a concrete countermeasure against the Parallel_PC_Oracle_CCA attack (Masking [13, 33]).

3.6.3 *Decryption-Failure (DF) Oracle-Based SCA.*

This category of attacks works by querying the decapsulation device with carefully perturbed ciphertexts, such that the decryption failures in the decrypted message m', depend upon the secret key. A side-channel oracle that is able to detect decryption failures can therefore recover the secret key.

The core idea of the attack is as follows: the attacker generates a valid ciphertext $ct = (\mathbf{u}, \mathbf{v})$ for a message m and adds single coefficient errors to the second component $\mathbf{v}$ (e.g.) $\bar{\mathbf{v}} = \mathbf{v} + e \cdot x^0$ (adding error to the first coefficient) where $e \in \mathbb{Z}^+$. This has the effect of perturbing the first coefficient of the erroneous message polynomial (i.e.) $\mathbf{m}'[0]$ by e (Line 27 of Decrypt in Alg.1), thereby increasing the first coefficient of the noise component $\mathbf{d}$ (i.e.) $\mathbf{d}[0]$ by e (Refer Eqn.3). If the error e is large enough to push $\mathbf{m}'[0]$ beyond $q/4$ (resp. $3q/4$) for $m'_0 = 0$ (resp. $m'_0 = 1$), then this flips m'_0 resulting in a decryption failure.

The size of e that triggers a decryption failure provides information about the original noise $\mathbf{d}[0]$, which is linearly dependent on the secret $\mathbf{s}$ (Eqn.3). Thus, an attacker who can obtain such information over several chosen ciphertexts can recover the full secret key [11, 30].

The first such attack exploiting a side-channel based DF oracle was proposed by Guo *et al.* [30] on Frodo KEM. They demonstrated that decryption failure can be detected through side-channel leakage from the ciphertext comparison operation (Line 21 in Alg.2). The key observation is that the re-computed ciphertext ct_R solely depends upon the decrypted message m' (Line 20 in Alg.2). Even a single bit change in m', results in a completely different recomputed ciphertext ct_R (due to the use of hash function). Thus, for a perturbed ciphertext ct which does not lead to a decryption failure, the ciphertext comparison only fails for a single coefficient of $\mathbf{v}$, while all other coefficients of both $\mathbf{u}$ and $\mathbf{v}$ match correctly, with that of the recomputed ciphertext. However, in case of a decryption failure, the coefficients of ct_R are completely random, which ensures that the ciphertext comparison fails in multiple coefficients with an overwhelming probability.

In this respect, Guo *et al.* [30] targeted the implementation of Frodo KEM, which utilizes a non-constant time comparison of the ciphertext comparison operation and exploited the difference in comparison time to instantiate a practical DF oracle, for full key recovery. Subsequently, Bhasin *et al.* [11] adapted the attack, which exploits power/EM side-channel leakage from constant-time implementations of the ciphertext comparison operation. They identified flaws in common approaches used for masking the ciphertext comparison operation proposed in [8, 51]. Subsequent works have proposed secure masking schemes for the ciphertext comparison operation used in lattice-based KEMs [11, 19]. We refer to these attacks using the label DF_Oracle_CCA attack. They have similar attack characteristic as that of the Binary_PC_Oracle_CCA attack (i.e.) (Communicate_DUT_IO, Profiled_Without_Clone, $\approx 5k - 6k$, Low_SNR), while consuming slightly more traces for key recovery, compared to the Binary_PC_Oracle_CCA attack.

Countermeasure: Masking the entire decapsulation procedure serves as a concrete countermeasure against the DF_Oracle_CCA attack (Masking [13, 33]).

3.6.4 *Full-Decryption (FD) Oracle-Based SCA.*

The aforementioned PC_Oracle_CCA and DF_Oracle_CCA attacks work by recovering anywhere between 1 to P bits of information about the secret key ($P \in \mathbb{Z}^+$), from a single chosen-ciphertext query. This gives rise to a natural question, whether it is possible to recover the entire message m' in a single query for chosen-ciphertexts. In this respect, Xu *et al.* [77] showed that operations that leak the complete message, exploited by message recovery attacks for valid ciphertexts, can also be exploited in a chosen-ciphertext setting to realize a full decryption (FD) oracle. In this manner, an attacker can recover 256 bits of information about the secret key $\mathbf{s}$ in a single trace.

In order to realize an FD oracle, they propose to construct ciphertexts $ct = (\mathbf{u}, \mathbf{v}) \in (R_q^k \times R_q)$ such that

$$\mathbf{u}_i = \begin{cases} U \cdot x^0 & \text{if } i = 0, \\ 0 & \text{if } 1 \leq i \leq k-1 \end{cases} \tag{12}$$

$$\mathbf{v} = V \cdot (\sum_{i=0}^{i=n-1} x^i) \tag{13}$$

where $(U, V) \in \mathbb{Z}^+$. The attacker can choose tuples (U, V) such that the decrypted message is nothing but

$$m'_i = \begin{cases} \mathcal{F}(\mathbf{s}_0[i]), & \text{if } 0 \leq i \leq n-1 \end{cases} \tag{14}$$

where every message bit m'_i is dependent upon the corresponding secret coefficient of $\mathbf{s}_0$ (i.e.) $\mathbf{s}_0[i]$. Moreover, attacker can choose (U, V) such that every message bit m'_i uniquely identifies the corresponding secret coefficient $\mathbf{s}_0[i]$ for $i \in [0, 255]$. In order to realize a practical FD oracle, Xu *et al.* [77] proposed to exploit leakage from the message encoding operation during re-encryption (Line 20 in Alg.1) which enables to recover the entire message in a single trace. Thus, full key recovery is possible in only 6 queries for Kyber512.

Similarly, Ravi *et al.* [58] and Ngo *et al.* [47, 48] showed that leakage from the message decoding operation (Line 28 in Alg.1) (i.e.) Message_Decode attack, can also be exploited in a chosen-ciphertext setting for key recovery, in approximately $6-20$ traces from schemes such as Kyber and Saber. We refer to these attacks using the label FD_Encode_Decode_Oracle_CCA. We define the attack characteristic using the tuple: (Communicate_DUT_IO, Profiled_Without_Clone, $\approx 6-20$, High_SNR).

Apart from attacks exploiting the power/EM side-channel, a few recent works have demonstrated FD oracle based key recovery attacks that exploit far-field amplitude modulated EM emanations from on-board antennas on mixed-signal chips [74, 75]. This side-channel can work over longer distances compared to the EM side-channel, but inherently contain more background noise, thereby increasing the number of traces for key recovery.

Targeting Protected Implementations of Message Encoding/Decoding Operation: We recall that in the presence of an Observe_DUT_IO attacker, Message_Encode and Message_Decode attacks targeting the message encoding and decoding procedures for message recovery, can be thwarted using the shuffling countermeasure (Shuffled_Encode, Shuffled_Decode). However, in the presence of a Communicate_DUT_IO attacker, when targeting the decapsulation procedure in a static key setting, Ravi *et al.* [58] showed that the shuffling countermeasures can be broken, exploiting the *ciphertext malleability* property of LWE/LWR-based schemes.

We briefly describe their attack exploiting leakage from the shuffled encoding operation, which recovers the message one bit at a time. The shuffling countermeasure does not remove leakage, but only ensures that the shuffling order of the message bits cannot be recovered by the attacker. Given a target ciphertext $ct = (\mathbf{u}, \mathbf{v})$ whose message is to be recovered, the attacker first submits the target ciphertext ct to the decapsulation procedure and recovers the individual message bits of m' through side-channels, and subsequently computes its Hamming Weight (HW). Subsequently, the attacker submits a perturbed ciphertext $ct' = (\mathbf{u}, \mathbf{v} + q/2 \cdot x^0)$ (i.e.) $q/2$ added to the first coefficient of $\mathbf{v}$. This has the effect of flipping the first message bit m'_0, resulting in a perturbed message m''. If $\mathsf{HW}(m'') = \mathsf{HW}(m') - 1$, then the perturbation flipped m'_0 from 1 to 0, thus deducing that $m'_0 = 1$. Otherwise if $\mathsf{HW}(m'') = \mathsf{HW}(m') + 1$, then $m'_0 = 0$. In this manner, an attacker can induce bit-flips in all the 256 bits of the message, to completely recover the message in 257 queries for Kyber KEM. Thus, shuffling increases the attacker's effort from recovering 256 bits in a single trace to recovering 1 bits per trace. Nevertheless, shuffling does not concretely prevent message recovery and key recovery in a chosen-ciphertext setting. Extending upon this idea, recently Ngo *et al.* [48] demonstrated improved attacks to break the combined shuffling and masking countermeasure for the message decoding operation in Saber.

We can clearly observe that an attacker with the Communicate_DUT_IO capability can perform improved attacks to break countermeasures such as shuffling, which are otherwise considered secure in the presence of an Observe_DUT_IO attacker. We refer to these attacks targeting the shuffled encoding/decoding procedure using the label Shuffled_Encode_Decode_FD_Oracle_CCA, and their characteristic tuple is (Communicate_DUT_IO, Profiled_Without_Clone, $\approx 2k - 3k$, High_SNR). The attack on the masked encoding/decoding procedure is denoted using the label Masked_Encode_Decode_FD_Oracle_CCA and its characteristic tuple is (Communicate_DUT_IO, Profiled_Without_Clone, $\approx 10 - 20$, High_SNR). The attack on the shuffled and masked encoding/decoding procedure using the label Shuffled_Masked_Encode_Decode_FD_Oracle_CCA. We define the attack characteristic using the tuple: (Communicate_DUT_IO, Profiled_Without_Clone, $\approx 2k - 3k$, High_SNR).

Countermeasure: As shown above, shuffled and masked implementations of the message encoding and decoding procedures do not prevent the realization of an FD oracle, for key recovery [47, 48, 58]. Since leakage from the message encoding/decoding procedure spans for only 1 to a few clock cycles for each message bit, the addition of jitter serves as a reasonable mitigation technique, but it does not concretely prevent the attack. Thus, increasing the key refreshment rate to repeatedly change the public key serves as the only strong countermeasure against the attack. This ensures that the attacker cannot obtain enough traces from the decapsulation procedure to recover a single secret key. However, the exact key refresh rate required to prevent these attacks depends upon the DUT and the attack setup.

3.6.5 Targeting NTT in a CCA setting. While leakage from the INTT instance over $\hat{g}' = (\hat{\mathbf{u}}' \circ \hat{\mathbf{s}})$ in the decryption procedure has been exploited for key recovery in the Observe_DUT_IO setting (Line 27 of Decrypt in Alg.1), the attack relies on extremely low-noise measurements for successful key recovery (SASCA_NTT attack [53, 55]). The authors show that the attack can tolerate a noise with standard deviation σ in the range $0.5 - 0.7$. Recently, Hamburg *et al.* [31] demonstrated that the sensitivity of these attacks to SNR can be significantly improved in a chosen-ciphertext setting (Observe_DUT_IO). Their idea was to craft chosen-ciphertexts such that coefficients of $\hat{g}'$ is sparse, and that leakage from the INTT operation over $\hat{g}'$ reportedly improves the effectiveness of the BP algorithm, by allowing more noise in the measurements, even when targeting masked implementations. They demonstrate a range of key recovery attacks with trace complexity ranging from k to $2k$ where k is the dimension of the module in Kyber KEM ($k = \{2, 3, 4\}$). The improved attack can tolerate much more noise with standard deviation $\sigma \leq 2.2$, thereby demonstrating significant improvement in SASCA_NTT attacks when performed in a chosen-ciphertext setting. We refer to the attacks targeting the NTT using the label CCA_SASCA_NTT attack. We define the attack characteristic using the tuple: (Communicate_DUT_IO, Profiled_With_Clone, 2 – 4, High_SNR).

Countermeasure: Shuffling or masking the NTT operation as proposed by Ravi *et al.* [63] provides concrete protection against SASCA style attacks.

Refer to Tab.1 for a tabulation of all side-channel attacks on the decapsulation procedure in the static key setting of Kyber KEM.

3.7 Protection Against SCA Assisted CCA

We observe that SCA-assisted CCA forms the largest category of attacks on Kyber KEM. Moreover, an attacker capable of querying the decapsulation device with chosen-ciphertexts can perform a variety of key recovery attacks, also capable of defeating certain masking and shuffling countermeasures [47–49, 58], with an incremental increase in attacker's effort compared to breaking unprotected implementations. Moreover, it is not clear which order of masking protection is required to achieve security in a given setting, especially given that the cost of masking significantly increases with the order of protection.

In this respect, we particularly focus on SCA-assisted CCA attacks which work with malicious ciphertexts, and present detection-based countermeasures, which test whether a received ciphertext is malicious. If detected as malicious, the DUT can simply reject the ciphertext and change/refresh the public-private key pair by re-running the key-generation procedure. This ensures that upon detection, further exposure of the secret key is prevented. In the following, we propose two *detection* countermeasures against the proposed CCAs for Kyber KEM.

3.7.1 Ciphertext Sanity Check. The main idea of this countermeasure stems from the observation that ciphertexts used for the Binary_PC_Oracle, Parallel_PC_Oracle, FD_Oracle and CCA_SASCA_NTT attacks are very sparse with several zero coefficients (Refer Eqn.5, 9 and 13 for the chosen-ciphertexts). However, the coefficients of a valid ciphertext are uniformly distributed in the range $[0, q]$, given that both ciphertext components are essentially LWE instances. This skew in the chosen ciphertexts can be easily detected and flagged as malicious ciphertexts before they can be decapsulated. While this countermeasure was also proposed by Xu *et al.* [77] to protect against attacks utilizing skewed ciphertexts, a concrete mathematical analysis and implementation of the same is not presented.

Detection Technique: In order to detect the skew in the ciphertexts, we chose to utilize the mean and standard deviation of the ciphertext coefficients. For a given polynomial $\mathbf{x} \in R_q$, we denote the mean (μ) and standard deviation (σ) of the coefficients of $\mathbf{x}$ as $\mu(\mathbf{x})$ and $\sigma(\mathbf{x})$ respectively. We performed empirical simulations to calculate the mean and standard deviation of $\mu(\mathbf{u})$ and $\sigma(\mathbf{u})$ for single polynomials of the ciphertext component $\mathbf{u}$, as well as $\mu(\mathbf{v})$ and $\sigma(\mathbf{v})$ for the ciphertext component $\mathbf{v}$, corresponding to valid ciphertexts of Kyber KEM. Refer below for the obtained values for the mean and standard deviation for all 4 of the statistical metrics for Kyber KEM.

$$\begin{aligned}(\mu(\mu(\mathbf{u})), \sigma(\mu(\mathbf{u}))) &= (1663, 60)\\ (\mu(\sigma(\mathbf{u})), \sigma(\sigma(\mathbf{u}))) &= (959, 27)\end{aligned} \quad (15) \qquad \begin{aligned}(\mu(\mu(\mathbf{v})), \sigma(\mu(\mathbf{v}))) &= (1560, 60)\\ (\mu(\sigma(\mathbf{v})), \sigma(\sigma(\mathbf{v}))) &= (957, 27)\end{aligned} \quad (16)$$

Based on the standard deviation σ for each of these metrics, the designer can choose an acceptable range for each of these 4 metrics. For example, if a tail length of $(6 \cdot \sigma)$ is chosen, then the acceptable range for $\mu(\mathbf{u})$ is $[\mu(\mu(\mathbf{u})) + 6 \cdot \sigma, \mu(\mu(\mathbf{u})) - 6 \cdot \sigma]$. The smaller the acceptable range, the higher the possibility of false positives (i.e.) detecting a valid ciphertext as malicious. However, a large acceptable range increases the chances of false negatives, thereby resulting in the acceptance of skewed malicious ciphertext as valid.

Evaluation: We deduced through empirical simulations that a tail of length (6σ) for both mean and standard deviation leads to a probability of $\approx 2^{-22}$ for rejection of a valid ciphertext. The rejection is done solely based on analyzing the size of the ciphertext coefficients, and this does not have any relation to the secret key. It is therefore trivial to observe that a false positive only hampers the performance of the scheme, but does not provide any additional information about the secret key. The implementor/designer can choose an appropriate range, based on the tolerance to allow false positives and rejection of valid ciphertexts. We henceforth refer to this as the CT_Sanity_Check countermeasure in this paper. One can also include other kinds of checks such as checking the number of zero coefficients in the received ciphertext as well as the decrypted message m', which can enhance confidence in the detection mechanism.

While this countermeasure is capable of detecting skewed ciphertexts, chosen-ciphertexts used in the DF_Oracle_CCA attack [11, 30] contain uniformly random coefficients. Thus, the DF_Oracle_CCA attack can bypass our CT_Sanity_Check countermeasure. In the following, we propose a novel countermeasure that is also capable of defeating CCA utilizing chosen ciphertexts with uniformly random coefficients.

3.7.2 *Message Polynomial Sanity Check.*

This countermeasure relies on analyzing the coefficients of the noisy message polynomial $\mathbf{m}' = (\mathbf{v}' - \mathbf{u}' \cdot \mathbf{s})$ obtained during decryption of the received ciphertext ct (Line 27 in Alg.1). For valid ciphertexts, we observe that the coefficients of the $\mathbf{m}'$ are distributed according to a very narrow Gaussian distribution near $q/2$ or 0 (i.e.) $\mathbf{m}[i] = q/2 \pm \delta$ for $m_i = 1$ and $\mathbf{m}[i] = 0 \pm \delta$ for $m_i = 0$

$$\mathbf{m}[i] = \begin{cases} q/2 \pm \delta & \text{if } m_i = 1, \\ 0 \pm \delta & \text{if } m_i = 0 \end{cases} \quad (17)$$

where $\delta \ll q \in \mathbb{Z}^+$. The span δ depends upon the distribution of the noise component $\mathbf{d}$ (Eqn.3). We performed empirical simulations to deduce the distribution of the coefficients of the noise component $\mathbf{d}$. They follow a Gaussian distribution with a standard deviation $\sigma = 79$, around 0 and $q/2$.

However, we observe that the distribution of the coefficients of $\mathbf{m}'$ is not maintained in the case of the DF_Oracle-based CCA attack. We observe that the DF_Oracle_CCA attack works by pushing one of the coefficients of $\mathbf{m}'$ ($\mathbf{m}'[i]$) to cross the $q/4$ threshold. This ensures that at least one message polynomial (i.e.) $\mathbf{m}'[i]$ is not within the expected range, corresponding to that of a valid ciphertext. This also applies to the following attacks which utilize malicious/hand-crafted ciphertexts: CCA_SASCA_NTT [31], Binary_PC_Oracle_CCA [22, 65, 68], Parallel_PC_Oracle_CCA [57, 71], DF_Oracle_CCA [11, 19, 30], FD_Oracle_CCA [58, 74, 75, 77] Masked_FD_Oracle_CCA [47, 49], Shuffled_FD_Oracle_CCA [58], Shuffled_Masked_FD_Oracle_CCA [48].

Detection Technique: Based on the aforementioned observation, we propose to test the distribution of the message polynomial coefficients for the received ciphertext. Let the acceptable range be $(q/2 \pm L \cdot \sigma)$ and $(0 \pm L \cdot \sigma)$ where $L \in \mathbb{Z}^+$ is left to the designer's choice. The larger the acceptable range $L \cdot \sigma$, the smaller the probability of flagging a valid ciphertext (false positive). However, choosing a smaller range raises the chances of missing detection of a malicious chosen ciphertext. Thus, it is important to choose a conservative value for L for improved security. Once an invalid ciphertext is detected, the corresponding secret key is discarded and a new one needs to be generated, for reasons that will be explained below.

Based on $\sigma = 79$ for the coefficients of the noise component $\mathbf{d}$ (Gaussian distribution), we also calculated that the probability of a false positive for detecting a valid ciphertext as malicious for Kyber KEM for $L = 6$ is $\approx\approx 7.129 \cdot 10^{-11} \approx 2^{-33}$. This false positive rate is very low for practical applications. We performed experimental simulations for $L = 6$, and we were not able to observe a false positive for more than 2^{25} valid decapsulations.

Evaluation: We subsequently tested several existing side-channel attacks [57, 58, 65] and found that for all attack ciphertexts used in these attacks there was a significant probability of triggering the countermeasure and thus discarding the secret key. More specifically, these attacks focus on one coefficient of the secret, and for all attack ciphertexts at least one possible value of this coefficient of the secret leads to the detection of the attack. The attack of Rajendran et al. [57] also includes an attack that targets multiple coefficients at once, but this improvement only increases the probability of triggering the countermeasure. We did not find parameter sets that reliably avoided our countermeasure. Thus we can conclude that for these attacks our countermeasure effectively restricts the number of useful invalid ciphertexts an attacker can input before the countermeasure is triggered and the secret key is discarded. The countermeasure would also effectively stop the attack described by Bhasin *et.al.* [11]. This attack relies on finding the boundary where the message bit is flipped, but due to the countermeasure, the region around the boundary results in the detection of the invalid ciphertext and the discard of the secret. Note that for $L = 6$ the discard region has approximately the same size as the accept region, making it infeasible to add an error to push the ciphertext towards the boundary without triggering the discard.

In-depth Analysis: The increased decryption failure probability makes the scheme more vulnerable to decryption failure attacks [20]. To mitigate this we only allow the adversary to obtain at most one failing ciphertext due to our countermeasure: if there is at least one coefficient outside this acceptable range, then we flag the ciphertext as invalid, discard the old public-private key pair and generate a new public-private key pair.

Allowing the adversary to obtain one failing ciphertext does not significantly impact security in this scenario. As can be seen from [18, 20, 21] one failing ciphertext is not enough to significantly reduce the security of the key pair, and the ciphertext is discarded after one failure caused by our countermeasure. Moreover, as the decryption failure probability is enlarged, the information in the decryption failure is reduced as discussed in [20]. This means that the leaked information from one failing ciphertext will be even smaller than in regular failure-boosting attacks.

More in-depth there are two scenarios to consider: first, the ciphertext is not accepted by the countermeasure, in which case the adversary has one failing ciphertext which as discussed previously does not significantly reduce the security of the public-private key pair. The key pair is subsequently discarded and as such the adversary can not gain additional information. Secondly, the ciphertext is accepted by the countermeasure, in which case there is no difference from the regular security framework of Kyber.

For a side-channel attacker, we observe that this countermeasure requires decrypting at least one chosen ciphertext for successful detection, however, the CCAs in interest require at least a few tens to thousand queries for key recovery. Thus, we argue that allowing a single decapsulation of the chosen ciphertext is not useful for the attacker. We henceforth refer to this countermeasure as Message_Poly_Sanity_Check throughout this paper. As we show later in Sec.4, this countermeasure can serve as a countermeasure for fault-assisted chosen-ciphertext attacks on Kyber KEM as well.

Comparison with Masking Countermeasures: The aforementioned detection countermeasures Message_Poly_Sanity_Check and CT_Sanity_Check) can be specifically used to protect against attacks against the decapsulation procedure in the chosen-ciphertext setting. As we show later in Sec.7.2, these countermeasures incur very less additional runtime compared to masking countermeasures for the decapsulation procedure. Thus, these countermeasures can be implemented as an add-on, on top of masked implementations of the decapsulation procedure. On the flip side, these countermeasures can only detect invalid/malicious ciphertexts, while they cannot deter attacks that work against CPA style attacks (NTT_Leakage_CPA) which work with valid ciphertexts.

4 FAULT-INJECTION ATTACKS ON KYBER KEM

In this section, we discuss reported fault attacks on Kyber KEM. For every FIA discussed in this paper, we also describe its characteristics based on the following parameters.

1. *Fault Injection Technique* (Attack_Vector): This characteristic denotes the type of fault injection technique used to carry out the attack - 1) Voltage/Clock Glitching (Glitching) 2) Laser Fault Injection (LFI) and 3) Electromagnetic Fault Injection (EMFI).
2. *Attacker's ability to communicate with DUT* (DUT_IO_Access): In this respect, we identify two categories: Observe_DUT_IO, Communicate_DUT_IO. Please refer to Sec.3.1.4 for the description of these categories.
3. *Targeted or Non-Targeted Fault* (Targeted_Or_Not): In this respect, we identify two categories:
    (a) Targeted_Fault: The attack works by injection faults to target specific variables or instructions, requiring to inject faults at a precise instance in time.
    (b) Non_Targeted_Fault: The attack does not require the injection of precise faults, and can work with random perturbations to the target computation. Thus, precise time synchronization is not required.
4. *Number of Faults within Single Computation* (Num_Faults): This characteristic denotes the number of faults to be injected within a single execution of the target procedure.
5. *Total number of Faulty Computations:* (Num_Executions): This indicates the total number of faulty computations/executions to recover the target secret variable. The number of executions is specified assuming that the expected fault is observed in every targeted execution of the computation. However, the exact number of faults required depends upon the design and the target platform.

Similar to SCA attacks on Kyber, we define the characteristic of each FIA on Kyber presented in the paper using the following tuple: (Injection_Technique, DUT_IO_Access, Targeted_Or_Not, Num_Faults, Num_Executions). In order

Table 1. Tablulation of reported SCA and their characteristics for the different procedures of Kyber KEM

Attack	Attack Characteristic					
	Attack_Vector	DUT_IO_Access	Profile_Requirement	No_Traces	SNR	Countermeasure
Key Generation						
SASCA_NTT [53, 55]	Power/EM	Observe_DUT_IO	Profiled_With_Clone	1	High_SNR	Shuffled_Masked_NTT
SASCA_KECCAK [37]	Power/EM	Observe_DUT_IO	Profiled_With_Clone	1	High_SNR	Shuffled_KECCAK
Encapsulation						
SASCA_NTT [53, 55]	Power/EM	Observe_DUT_IO	Profiled_With_Clone	1	High_SNR	Shuffled_Masked_NTT
SASCA_KECCAK [37]	Power/EM	Observe_DUT_IO	Profiled_With_Clone	1	High_SNR	Shuffled_KECCAK
Message_Encode [4, 69, 77]	Power/EM	Observe_DUT_IO	Profiled_With_Clone	1	High_SNR	Shuffled_Encode
		Communicate_DUT_IO	Profiled_Without_Clone	1	High_SNR	Shuffled_Encode
Masked_Message_Encode [47, 49]	Power/EM	Observe_DUT_IO	Profiled_With_Clone	1	High_SNR	Shuffled_Encode
		Communicate_DUT_IO	Profiled_Without_Clone	1	High_SNR	Shuffled_Encode
Decapsulation (Ephemeral Key)						
SASCA_NTT [53, 55]	Power/EM	Observe_DUT_IO	Profiled_With_Clone	1	High_SNR	Shuffled_Masked_NTT
		Communicate_DUT_IO	Profiled_Without_Clone	1	High_SNR	Shuffled_Masked_NTT
SASCA_KECCAK [37]	Power/EM	Observe_DUT_IO	Profiled_With_Clone	1	High_SNR	Shuffled_KECCAK
		Communicate_DUT_IO	Profiled_Without_Clone	1	High_SNR	Shuffled_KECCAK
Message_Decode [58]	Power/EM	Communicate_DUT_IO	Profiled_Without_Clone	1	High_SNR	Shuffled_Decode
		Observe_DUT_IO	Profiled_With_Clone	1	High_SNR	Shuffled_Decode
Masked_Message_Decode [47, 49]	Power/EM	Communicate_DUT_IO	Profiled_Without_Clone	1	High_SNR	Shuffled_Decode
		Observe_DUT_IO	Profiled_With_Clone	1	High_SNR	Shuffled_Decode
Decapsulation (Static Key)						
CT_Sanity_Check Message_Poly_Sanity_Check NTT_Leakage_CPA [16, 45]	Power/EM	Observe_DUT_IO	Non_Profiled	≈ 200	Low_SNR	Masking
CCA_SASCA_NTT [31]	Power/EM	Communicate_DUT_IO	Profiled_With_Clone	$2-4$	Low_SNR	Shuffled_Masked_NTT, CT_Sanity_Check, Message_Poly_Sanity_Check
Binary_PC_Oracle_CCA [22, 65, 68]	Power/EM [65, 68], Timing [22]	Communicate_DUT_IO	Profiled_Without_Clone	$\approx 2k-3k$	Low_SNR	Masking, CT_Sanity_Check, Message_Poly_Sanity_Check
Parallel_PC_Oracle_CCA [57, 71]	Power/EM	Communicate_DUT_IO	Profiled_Without_Clone	$\approx 100-200$	Low_SNR	Masking, CT_Sanity_Check, Message_Poly_Sanity_Check
		Communicate_DUT_IO	Profiled_With_Clone	$\approx 300-500$	Low_SNR	Masking, CT_Sanity_Check, Message_Poly_Sanity_Check
DF_Oracle_CCA [11, 19, 30]	Power/EM [11, 19], Timing [30]	Communicate_DUT_IO	Profiled_Without_Clone	$5k-7k$	Low_SNR	Masking, CT_Sanity_Check, Message_Poly_Sanity_Check
FD_Oracle_CCA [58, 74, 75, 77]	Power/EM [58, 77], Ampliute Modulated EM [74, 75]	Communicate_DUT_IO	Profiled_Without_Clone	$6-20$	High_SNR	CT_Sanity_Check, Message_Poly_Sanity_Check
Masked_FD_Oracle_CCA [47, 49]	Power/EM	Communicate_DUT_IO	Profiled_Without_Clone	$6-20$	High_SNR	CT_Sanity_Check, Message_Poly_Sanity_Check
Shuffled_FD_Oracle_CCA [58]	Power/EM	Communicate_DUT_IO	Profiled_Without_Clone	$\approx 1k-3k$	High_SNR	CT_Sanity_Check, Message_Poly_Sanity_Check
Shuffled_Masked_FD_Oracle_CCA [48]	Power/EM	Communicate_DUT_IO	Profiled_Without_Clone	$\approx 1k-3k$	High_SNR	CT_Sanity_Check, Message_Poly_Sanity_Check

to explain the different attacks, we utilize the algorithm of IND-CPA secure PKE of Kyber in Alg.1 and algorithm of IND-CCA secure Kyber KEM in Alg.2. We also refer the reader to Fig.2 for an example key-exchange protocol that can be built using IND-CCA secure Kyber KEM.

4.1 FIA on Key Generation

The key generation procedure serves as an attractive target for an attacker, particularly in an ephemeral key setting, since it is performed for every new key exchange. Injection of faults in the key-generation procedure could lead to faulty public keys that could easily compromise the secret key.

4.1.1 Targeting Sampling of Secrets. In this respect, Ravi *et al.* [64] proposed the first practical fault attack targeting the sampling of secrets and errors to generate LWE instances. Their attack stems from the observation that the seed used to sample the secret $\mathbf{s}$ and errors $\mathbf{e}$ only differ by a single byte (i.e.) $seed_B$ appended by single-byte nonces $coins_s$ and $coins_e$ (Lines 5-6 of KeyGen in Alg.1). Thus, the attacker can use faults to force nonce reuse (i.e.) $coins_s = coins_e$. This creates LWE instances of the form, $\mathbf{t} = \mathbf{A} \cdot \mathbf{s} + \mathbf{s} = (\mathbf{A} + \mathbf{I}) \cdot \mathbf{s}$, that can be trivially solved using Gaussian elimination. Thus, the faulty public keys can be directly solved to recover the secret key. The faulty public keys are still valid to be used for valid key exchange, and the injected faults have only reduced the entropy of the secret key. The authors demonstrated the practicality of nonce-reuse using Electromagnetic Fault Injection (EMFI) on the ARM Cortex-M4 microcontroller. The attack requires to inject multiple targeted faults on the nonces used during the sampling procedure $(1 - 10)$ depending upon the target scheme to attack, for full key recovery. We refer to this attack using the label Nonce_Fault attack. We describe the attack characteristic using the following tuple: (EMFI, Observe_DUT_IO, Targeted_Fault, 4−8, 1).

Countermeasure (Our Proposal): We propose to implement a dedicated verification procedure, which checks for equality of polynomials in the secret $\mathbf{s} \in R_q^k$ and error $\mathbf{e} \in R_q^k$. Firstly, polynomials within the same module $\mathbf{s} \in R_q^k$ and $\mathbf{e} \in R_q^k$ are checked for equality. Instead of comparing all the coefficients, a set of X coefficients is picked at random for checking equality and X is large enough such that the probability of all X corresponding coefficients having the same value is very low. For Kyber768 with coefficients in $[-2, 2]$ (distributed based on CBD), the probability of X pairs of coefficients having the same value is $\approx 2 \cdot 10^{-6}$. This is the false positive rate for $X = 10$. The designer can choose an appropriate value for X based on an acceptable false positive rate. The same comparison is also done between polynomials of $\mathbf{s}$ and $\mathbf{e}$.

The aforementioned scheme is implemented as follows: Firstly, a random value $rand \in \mathbb{Z}^+$ is sampled. Let the result of the verification procedure be denoted as $verify\_result$, which is initialized as $rand$. For every pair of polynomials which is found to be equal, $result \in \mathbb{Z}^+$ is incremented by $w \in \mathbb{Z}^+$. To incorporate redundancy, this check and increment can be done $y \in \mathbb{Z}^+$ times. Thus, if any two polynomials of $\mathbf{s}$ or $\mathbf{e}$ are found to be equal, the value of $verify\_result$ is incremented by $y \cdot w$. If no pair of polynomials are equal, then ($verify\_result = rand$), indicating the success of verification. If this is not the case, the verification has failed. We denote this dedicated verification procedure as Verify_Equality.

However, one can argue that the countermeasure can be defeated by simply skipping Verify_Equality. Such double fault attacks can be prevented by carefully designing a loop counter, that can detect such trivial skipping fault attacks. We propose to incorporate a Dynamic_Loop_*Counter* protection for the Verify_Equality procedure in the following manner, so as to prevent against trivial double fault attacks.

Let the total number of coefficients of all polynomials of $\mathbf{s}$ and $\mathbf{e}$ to be compared be denoted as C. First, a random non-zero integer $g \in \mathbb{Z}^+$ is sampled. Then, a loop counter lc is initialized to 0 and its value is increased by g for every coefficient that is compared (i.e.) for every coefficient comparison. The public key pk is generated and stored in a temporary variable $temp$. It is copied one byte at a time to the actual output variable that is considered the public key pk_{out} (randomly initialized), only if the loop counter value is equal to the expected value ($lc = C \cdot g$), indicating completion of the verification procedure and ($verify\_result = rand$), indicating the success of the verification procedure. This ensures that the correct public key is generated at the output, only when the verification procedure has passed, and has been fully executed. This comparison is done for every byte moved from $temp$ to pk_{out}. One can also augment with checks for a non-zero value for lc and $verify\_result$ to prevent zeroization attacks.

The aforemetioned two level protection strategy of combining Verify_Equality and Dynamic_Loop_*Counter* protection is together referred to as Verify_Nonce_Fault countermeasure. Please refer to Alg.5 for the pseudo-code of the Verify_Nonce_Fault protection.

We argue that Verify_Nonce_Fault provides improved resistance against fault attacks in the following manner:

(1) Zeroization of variables such as g (Line 3 in Alg.5) or $rand$ (Line 3) cannot pass verification in Line 11, thereby generating a random public key, which is unuseful for an attacker.
(2) Skipping the Verify_Equality procedure (Line 8) also ensures that the loop counter verification fails (Line 11), thereby offering protection.
(3) The value of lc and $verify\_result$ change for every execution, and thus injection of precise faults on these variables to force successful comparison is challenging to achieve in practice (Line 11). Even if the attacker can force a successful comparison, this has to be repeated for a few thousand bytes of the public key pk for Kyber KEM, which is also challenging to achieve in practice (Line 11).

Thus, we argue that it is possible to design the protection such that the implementation is not susceptible to trivial fault attacks, and that the attacker requires to inject several highly synchronized faults to bypass the Verify_Nonce_Fault protection.

Algorithm 5: Verify_Nonce_Fault countermeasure for KeyGen of Kyber KEM

1: **procedure** Verify_Nonce_Fault Protected KeyGen
2: $lc := 0$
3: $g \in \mathbb{Z}^{+} \leftarrow$ Sample_Random()
4: $expected\_lc := g \cdot C$ ▷ C number of operations to be accounted for in Verify_Equality procedure
5: $rand \in \mathbb{Z}^{+} \leftarrow$ Sample_Random()
6: $verify\_result = rand$ ▷ Initializing result of Verify_Equality
7: Sample secret $\mathbf{s} \in R_q^k$ and error $\mathbf{e} \in R_q^k$
8: $verify\_result =$ Verify_Equality($\mathbf{s}, \mathbf{e}, lc, g$) ▷ If Verify_Equality fails, $verify\_result\ != \ rand$
9: $tmp =$ Compute_Public_Key() ▷ Compute pk and store in tmp array
10: **for** j from 0 to $(nb - 1)$ **do** ▷ Copy nb bytes of public key from tmp to pk_{out}
11: **if** $(expected\_lc == lc)$ and $(expected\_lc\ != \ 0)$ and $(verify\_result == rand)$ and $(verify\_result != 0)$ **then**
12: $pk_{out}[j] = tmp$ ▷ Copy the public key byte if verification passes...
13: **else**
14: $pk_{out}[j] = rand()$ ▷ Copy a random byte if verification passes...
15: **end if**
16: **end for**
17: **end procedure**

4.1.2 Targeting NTT. Ravi *et al.* [66] proposed a novel fault attack targeting the NTT operation. They identified a single point of failure, that can be targeted through faults, to *zeroize all twiddle factors* used within the NTT. When this is targeted on the NTT over secrets or errors, it can severely reduce the entropy of the secret/error. This results in faulty yet valid LWE instances, which easily compromise the secret key. For instance, let the DUT compute NTT over the polynomial $\mathbf{x} = (\mathbf{x}_0, \mathbf{x}_1, \ldots, \mathbf{x}_{n-1})$. If the twiddle factors used in the NTT computation are zeroized, then the resulting faulty NTT output is $\hat{\mathbf{x}}^* = (\mathbf{x}_0, 0, \ldots, 0)$. If $\hat{\mathbf{x}}^*$ is used in subsequent computations, then the faulty polynomial $\mathbf{x}^*$ is nothing but $\mathbf{x}^* = (\mathbf{x}_0, \mathbf{x}_0, \ldots, \mathbf{x}_0)$. Thus, the injected fault has effectively reduced the entropy of $\mathbf{x}$ to just a single coefficient $\mathbf{x}_0$, which can be easily guessed by an attacker, given the short span of secrets and errors used in Kyber KEM.

The authors studied the assembly-optimized implementation of NTT for Kyber and Dilithium, on the ARM Cortex-M4 device, available in the *pqm4* library [38]. They showed that a single targeted fault using EMFI on the address pointer to the twiddle factor array can be used to effectively zeroize all the twiddle factors. In this respect, an attacker can target the NTT over the secret **s** or error **e** in the key-generation procedure (Line 5, 6 in Alg.1). This results in the utilization of low-entropy secrets and errors to generate a faulty public key, which can be easily solved to recover the secret key. The same faulty secret key is also used within the decapsulation procedure, as Kyber saves the secret key in the NTT domain. This therefore ensures the correctness of Kyber KEM, even upon injection of fault in the NTT, only during key generation. We refer to this attack using the label NTT_Twiddle_Fault attack. The attack characteristic can be defined using the tuple: (EMFI, Observe_DUT_IO, Targeted_Fault, 1, 1).

Countermeasure: Ravi *et al.* [66] proposed a few detection-based countermeasures to test zeroization of the twiddle constants, before utilization for the NTT operation. One can also adopt a small testing procedure to check if the twiddle factors to be used for the NTT, all have a non-zero value. If one or more twiddle constants have a zero value, then the entire procedure can be aborted. As an additional protection, one can also test the entropy of the NTT output, given that the faulty NTT output has a very low entropy with a single non-zero coefficient. We refer to these detection countermeasures using the label NTT_Twiddle_Check.

Refer to Tab.2 for a tabulation of all fault-injection attacks on the key-generation procedure of Kyber KEM.

4.2 FIA on Encapsulation

The fault attacks applicable to the key generation procedure (i.e.) Nonce_Fault and NTT_Twiddle_Fault attack are also applicable to the encapsulation procedure. The Nonce_Fault attack on the encapsulation procedure can be done by targeting the nonces used to sample the ephemeral secret **r** (Line 14 in Alg.1), whose knowledge can be used to perform message recovery. Similarly, the NTT_Twiddle_Fault attack can be mounted by targeting the NTT operation over **r** (Line 17), which reduces the entropy of **r** resulting in message recovery. Thus, the attacks over the key-generation procedure apply in the same manner to the encapsulation procedure of Kyber KEM.

Countermeasure: The Verify_Nonce_Fault and NTT_Twiddle_Check serve as concrete countermeasures against the aforementioned attacks on the encapsulation procedure of Kyber KEM.

Refer to Tab.2 for a tabulation of all fault-injection attacks on the encapsulation procedure of Kyber KEM.

4.3 FIA on Decapsulation

With respect to FIA on the decapsulation procedure, we consider two scenarios. In the case of ephemeral key setting, faulting the decapsulation procedure does not provide any information about the secret key or the message. The attacker can only inject faults to corrupt the decapsulation of valid ciphertexts, which amounts to a Denial of Service (DoS) attack. However, in the case of the static key setting, a Communicate_DUT_IO attacker can query the DUT with chosen-ciphertexts and the result of corresponding faulty decapsulations can potentially recover the long-term secret key. The following are different fault attacks reported on the decapsulation procedure.

4.3.1 Targeting Ciphertext Equality Check. One obvious target within the decapsulation procedure is to simply skip the final ciphertext comparison operation, whose result indicates the validity of the ciphertext (Line 21 in Alg.2). An attacker who can skip the equality check for his/her chosen ciphertexts effectively reduces the security from IND-CCA security to IND-CPA security. Skipping the equality check ensures that the session key K contains critical information about the decrypted message m', even for the attacker's chosen ciphertexts. This knowledge of the session keys, for several such chosen ciphertexts leads to recovery of the long-term secret key $\mathbf{s}$.

Recently, Xagawa *et al.* [76] surveyed optimized software implementations of several PQC schemes on the ARM Cortex-M4 microcontroller and identified that implementations of several schemes including Kyber KEM are vulnerable to trivial skipping fault attacks. The ciphertext equality check within the optimized software implementation of Kyber KEM from the *pqm4* library [38], is done in the following manner. The pre-key $\bar{K}'$ is computed using m' and pk after decryption (Line 18 in Alg.2), and is stored in an array T (Line 19). If ciphertext comparison fails (invalid/malicious ciphertext), a pseudo-random value z is written into T using a conditional move operation (Line 22). Else, the pre-key in the array T is not overwritten. Then, T is used to derive the final shared session key K (Line 24).

The vulnerability is that the decapsulation procedure writes the sensitive pre-key $\bar{K}'$ onto T (assuming successful decapsulation), before checking the validity of the ciphertext. Thus, simply skipping the subsequent conditional move operation (Line 22) for malicious ciphertexts, ensures that $\bar{K}'$ is used to generate the shared session key K instead of the pseudo-random z, even for invalid ciphertexts. Xagawa *et al.* [76] exploited this vulnerability through simple clock glitches and could subsequently recover the secret key in a few thousand chosen-ciphertext queries, similar to the Binary_PC_Oracle_CCA attack [22, 65]. We refer to this attack using the label Skip_CT_Compare. The attack characteristic can be defined by the tuple: (Glitching, Communicate_DUT_IO, Targeted_Fault, 1, $1k - 3k$).

Countermeasure (Our Proposal): We propose two levels of protection for the ciphertext comparison operation, targeted by the Skip_CT_Compare attack. Trivial skipping of the entire ciphertext comparison operation (Line 21 in Alg.2) can be detected through the Dynamic_Loop_*Counter* protection (Refer Sec.4.1.1). As a second level of protection, we propose to remove the vulnerability that allows for trivial skipping attacks. We alter the conditional move operation (Line 22 in Alg.2) in the following manner. We ensure that the pre-key $\bar{K}'$ is written into a temporary variable tmp (initialized with a random value). Subsequently, the tmp variable containing the pre-key is copied into the array T, one byte at a time, only if both the following conditions are satisfied - 1) ciphertext comparison succeeds and Dynamic_Loop_*Counter* verification passes. Both these checks are done for every byte that is copied from tmp to T (32 bytes). If either of the conditions fails, then the pseudo-random value z is copied into T. We refer to this two-stage protection using the label Protect_CT_Compare. The implementation of this countermeasure can be done in a similar manner, as that of the Verify_Nonce_Fault countermeasure, and we thus refer the reader to Sec.4.1.1 for more details on the implementation and effectiveness of the countermeasure.

4.3.2 *Ineffective Fault Analysis.*

Pessl and Prokop [54] recently proposed a novel ineffective fault attack against the decapsulation procedure. It works by injecting targeted faults within the message decoding operation during decryption (Line 28 in Alg.1), such that the resulting success/failure of decapsulation can be used to infer critical information about the secret key.

We briefly describe the main idea of their attack. The attacker constructs a valid ciphertext ct and submits ct for decapsulation by the DUT. Subsequently, a targeted fault is injected to skip the addition operation during decoding of a message polynomial coefficient $\mathbf{m}'[i]$ into the message bit m'_i (Refer to the code snippet of the message decoding procedure in Fig.1). The injected fault results in a flip of m'_i (decapsulation failure), only if the corresponding coefficient of the noise component $\mathbf{d}[i] < 0$ (Refer Eqn.3). However, there is no change in m'_i when $\mathbf{d}[i] \geq 0$ (decapsulation success).

Thus, the knowledge of whether the injected fault resulted in a decapsulation success/failure helps infer information about $\mathbf{d}[i]$, which is linearly dependent upon the secret key $\mathbf{s}$. This can be done for several valid ciphertexts to fully recover the secret key in $6.5k - 13k$ queries for Kyber KEM. However, the number of queries for key recovery can be reduced to $5k - 7k$ using improved post-processing techniques as shown in [34]. In essence, the attack utilizes fault injection to realize a practical decryption failure (DF) oracle for valid ciphertexts, for key recovery. The attack was demonstrated using clock glitching on the ARM Cortex-M4 microcontroller and requires injecting a targeted skipping fault in the message decoding procedure. We refer to this attack using the label Ineffective_FIA. The attack characteristic can be described using the tuple: (Glitching, Communicate_DUT_IO, Targeted_Fault, 1, $5k - 7k$).

Countermeasure: Since the attack specifically targets the message decoding procedure, simply shuffling the message decoding procedure (i.e.) Shuffled_Decode serves as a concrete countermeasure against the attack.

4.3.3 Fault Correction Attack. Hermelink *et al.* [34] proposed a novel fault attack on the decapsulation procedure, which adopts a slightly different approach. The attacker constructs a valid ciphertext $ct = (\mathbf{u}, \mathbf{v})$, and adds a single-bit perturbation of $\approx q/4$ to one of the coefficients of $\mathbf{v}$ (i.e.) $\mathbf{v}[i]$. This perturbed ciphertext $ct' = (\mathbf{u}', \mathbf{v}')$ is submitted to the DUT for decapsulation. Upon submitting the perturbed ciphertext, a fault is injected anytime after decryption (Line 17 in Alg.2) and before ciphertext comparison (Line 21), to correct the single-bit perturbation in the ciphertext stored in memory. If the introduced perturbation resulted in correct decryption, then the injected fault corrects the single-bit perturbation in the ciphertext, ensuring successful decapsulation. However, if the initial perturbation resulted in a decryption failure ($\mathbf{d}[i] < 0$), then it results in decapsulation failure, even after correcting the perturbation in the stored ciphertext through faults, since all the ciphertext coefficients of ct_R are uniformly randomized during re-encryption. This information obtained about $\mathbf{d}$ over $5k - 7k$ such queries can recover the full secret key.

Unlike the attack of Pessl and Prokop, the attack of Hermelink *et al.* [34] does not have any timing constraints for fault injection, as it only needs to inject a bit-flip fault in memory, anytime between the decryption and ciphertext comparison operation. However, injecting precise single bit-flip faults in memory requires detailed information about the target device as well as the implementation, and an extensive profiling of the target device. The attack characteristic can be defined by the following tuple: (LFI, Communicate_DUT_IO, Targeted_Fault, 1, $5k - 7k$).

More recently, Delvaux [23] improved the attack of Hermelink *et al.* [34] by expanding the attack surface to several operations within the decapsulation procedure, while also working with a variety of more relaxed fault models such as arbitrary bit flips, set-to-0 faults, random faults, and instruction skip faults. However, attacks relying on a relaxed fault model could require about $100k$ chosen-ciphertext queries for full key recovery, depending upon the practicality of the fault model. The attack characteristic can be defined by the following tuple: (Glitching, Communicate_DUT_IO, Targeted_Fault, 1, $10k - 100k$). We refer to the aforementioned attacks using the label Fault_Correction attack.

Countermeasure (Our proposal): We observe that the attack works with perturbed ciphertexts, and observe that the corresponding coefficients of the erroneous message polynomial $\mathbf{m}'$ upon decryption do not satisfy the distribution of the message polynomial of a valid ciphertext. Thus, our proposed Message_Poly_Sanity_Check serves as a concrete detection countermeasure against the attack.

Refer to Tab.2 for a tabulation of all fault-injection attacks on the decapsulation procedure of Kyber KEM in the static key setting.

Table 2. Tablulation of reported FIA and their characteristics for the different procedures of Kyber KEM

Attack	Attack Characteristic					
	Attack_Vector	DUT_IO_Access	Targeted_Or_Not	Num_Faults	Num_Executions	Countermeasure
Key Generation						
Nonce_Fault [64]	EMFI	Observe_DUT_IO	Targeted_Fault	$4-8$	1	Verify_Nonce_Fault
NTT_Twiddle_Fault [66]	EMFI	Observe_DUT_IO	Targeted_Fault	1	1	NTT_Twiddle_Check
Encapsulation						
Nonce_Fault [64]	EMFI	Observe_DUT_IO	Targeted_Fault	$4-8$	1	Verify_Nonce_Fault
NTT_Twiddle_Fault [66]	EMFI	Observe_DUT_IO	Targeted_Fault	1	1	NTT_Twiddle_Check
Decapsulation (Static Key)						
Skip_CT_Compare [76]	Glitching	Communicate_DUT_IO	Targeted_Fault	1	$1k-3k$	Protect_CT_Compare
Ineffective_FIA [54]	Glitching	Communicate_DUT_IO	Targeted_Fault	1	$5k-7k$	Shuffled_Decode
Fault_Correction [23, 34]	LFI [34]	Communicate_DUT_IO	Targeted_Fault	1	$5k-7k$	Message_Poly_Sanity_Check
	Glitching [23]	Communicate_DUT_IO	Targeted_Fault	1	$10k-100k$	

5 FAULT-INJECTION ATTACKS ON DILITHIUM

In this section, we discuss fault attacks that are applicable to the Dilithium signature scheme. We utilize the same characteristics to describe FIA on Dilithium, that were used to describe FIA on Kyber (Refer Sec.4). We utilize the algorithm of the Dilithium signature scheme in Alg.3-4 to explain the different attacks. We note that the secret key sk of Dilithium has multiple components: $sk = (seed_A, K, tr, \mathbf{s_1}, \mathbf{s_2}, \mathbf{t_0})$ (Line 9 in Alg.3). Among them, we refer to $\mathbf{s}_1$ as the primary secret, since the knowledge of $\mathbf{s}_1$ is sufficient to forge signatures of Dilithium for any chosen message, as shown in [15, 62].

5.1 FIA on Key Generation

The key generation procedure of Dilithium can serve as an attractive target for fault attacks when the application utilizes self-signed certificates, where key generation is performed on the DUT. In this scenario, the following attacks are applicable to the key generation procedure.

*5.1.1 Targeting Sampling of Secrets (*Nonce_Fault*).* The polynomials of the secret $\mathbf{s}_1$ and $\mathbf{s}_2$ of Dilithium are sampled using the same seed $seed_S$, but with different delimiters/nonces (Line 3 of Sign in Alg.3). Ravi *et al.* [64] showed that an attacker can force nonce reuse through faults to generate weak LWE instances, which can be potentially solved to recover the secret key. Dilithium utilizes rounding of the public key (Line 7), which poses an additional challenge for the attacker to recover the secret key. Nevertheless, the induced nonce reuse through faults significantly reduces the security of the public keys, as the full public key can be reconstructed by observing several valid signatures. The attack characteristic is defined using the tuple: (EMFI, Observe_DUT_IO, Targeted_Fault, $8-15$, 1).

Countermeasure (Our Proposal): The Verify_Nonce_Fault countermeasure (Sec.4.1.1) can serve as a concrete protection against the Nonce_Fault attack.

*5.1.2 Targeting NTT (*NTT_Twiddle_Fault*).* Ravi *et al.* [66] proposed to target the NTT instances through the NTT_Twiddle_Fault attack, in the key generation procedure of Kyber KEM, to create faulty yet valid secret keys with very low entropy. While NTT is also computed over the secret key component $\mathbf{s}_1$ in Dilithium, the fault attack is not applicable to the key generation procedure of Dilithium. This is because the faulty NTT transformed version of the secret $\mathbf{s}_1$ is only used to generate the LWE instance (i.e.) public key (Line 6 in Alg.3). The key-generation procedure however saves the original secret $\mathbf{s}_1$ in the normal domain, as the secret key. Thus, the signing procedure performs a fresh NTT computation over the secret $\mathbf{s}_1$ while generating signatures. This violates the correctness of the generated signatures, thereby rendering the attack on the key generation procedure of Dilithium useless.

Refer to Tab.3 for a tabulation of all fault-injection attacks on the key-generation procedure of the Dilithium signature scheme.

5.2 FIA on Signing Procedure

The signing procedure of Dilithium remains the main target of fault injection attacks, as the signing procedure utilizes the long-term secret key *sk* to generate multiple signatures, given the long lifetime of the key pairs used in signature schemes. The following attacks are applicable to the signing procedure of Dilithium.

5.2.1 Injecting Random Faults on the Secret Key. Bindel *et al.* [12] reported the first fault vulnerability analysis of lattice-based signature schemes such as GLP [29] and BLISS [24], 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 $\approx 1k - 2k$ faulty signatures, to obtain knowledge about the originally perturbed coefficients of $\mathbf{s}_1$, one at a time to fully recover $\mathbf{s}_1$.

Along the same lines, Islam *et al.* [36] 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. The attack does not require communication with the signing DUT, and can work on both the deterministic and probabilistic variants of Dilithium. The attack characteristic is defined using the tuple: (EMFI, Observe_DUT_IO, Targeted_Fault, 1, $\approx 1k - 2k$).

Countermeasure: 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 using the label Verify_After_Sign. While this countermeasure has been proposed by several works [12, 15], its concrete implementation and performance evaluation has not been studied by prior works.

5.2.2 Generic Differential Fault Analysis (DFA). Bruinderink and Pessl [15] 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 (Communicate_DUT_IO), and submits a signature query for a randomly chosen message m. Let the primary signature component be $\mathbf{z} = \mathbf{s}_1 \cdot c + \mathbf{y}$ (Line 27 in Alg.3). The attacker again 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 only a single random fault (using glitches) anywhere within 68% of the execution time of a single iteration of the signing procedure can result in full key recovery, thereby demonstrating the effectiveness of their attack. Referring to the signing procedure in Alg.3, the random fault can be injected anywhere in lines 12 and 23-27. We henceforth refer to this attack as the Generic_DFA attack on Dilithium. Since the attack is a DFA style attack, it can only work on the deterministic variant of Dilithium, but not on the probabilistic variant. The attack characteristic can be defined by the following tuple: (Glitching, Communicate_DUT_IO, Non_Targeted_Fault, 1, 1).

Countermeasure: 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 [15] 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. However, converting the signing procedure to being probabilistic, also serves as a concrete countermeasure against the attack.

5.2.3 Loop Abort Fault Attack. Espitau *et al.* [26] proposed a novel fault attack to directly target the nonce $\mathbf{y}$ in Fiat-Shamir abort-based signature schemes such as GLP signature scheme [29] and BLISS [24]. They proposed to use faults to prematurely abort the loop, that samples the single coefficients of $\mathbf{y}$ (Line 22 in Alg.3), thereby resulting in the generation of nonces with low degrees. In other words, by skipping the loop that samples individual coefficients of $\mathbf{y}$, one can ensure that the remaining coefficients of $\mathbf{y}$ are unsampled, and there is a high chance that these unsampled coefficients have a value of 0. If so, the faulted signature $\mathbf{z}$ contains several coefficients which are nothing but the unmasked coefficients of the product $\mathbf{s}_1 \cdot \mathbf{c}$ (Line 27 in Alg.3). The authors show that a single targeted fault in the sampling procedure of $\mathbf{y}$ can result in full key recovery. Though this attack was only demonstrated on the GLP signature scheme [29], this attack can potentially be applied to Dilithium for full key recovery. Since this attack does not involve differential analysis, it is therefore applicable to both the probabilistic and deterministic variants of Dilithium. We refer to this attack using the label Loop_Abort_Fault. Its characteristic can be defined using the tuple: (Glitching, Observe_DUT_IO, Targeted_Fault, 1, 1).

Countermeasure (Our Proposal): We propose a two-level protection mechanism, similar to that of the Verify_Nonce_Fault (Sec.4.1.1), which works in the following manner. Firstly, we utilize the Dynamic_Loop_*Counter* protection to keep track of the number of sampled coefficients of $\mathbf{y}$. 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 comparison succeeds, and this comparison is done for every byte copied from *temp* to *sig*. We refer to this two-level countermeasure using the label Verify_Loop_Abort. We refer to Sec.4.1.1 for more details on the implementation and effectiveness of the countermeasure.

5.2.4 Skip Addition Attack. Bindel *et al.* [12] proposed theoretical skipping fault attacks targeting the final addition operation used to generate $\mathbf{z}$ (Line 27 in Alg.3). Skipping the addition of $\mathbf{y}$ with the product $(\mathbf{s}_1 \cdot c)$, 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.* [62] proposed a more subtle fault attack on the deterministic variant of Dilithium, which involves skipping of the addition operation for single coefficients of $\mathbf{z}$ (Line 27 in Alg.3). An attacker can then use a DFA technique similar to [15], to recover the secret module $\mathbf{s}_1$ in $\approx 1k - 2k$ such faulty signatures. While the attack has only been demonstrated on the deterministic variant of Dilithium, its applicability to the probabilistic variant is not clear and is yet to be studied. We refer to these attacks as the Skip_Addition fault attacks, whose characteristic can be defined using the tuple: (EMFI, Communicate_DUT_IO, Targeted_Fault, 1, $\approx 1k - 2k$).

Countermeasure: The use of a Verify_Loop_Abort like countermeasure can be used to detect skipping of any of the addition operations to generate the primary signature component $\mathbf{z}$. However, the protection does not defeat attacks that skip the addition of single coefficients through corruption of underlying assembly instructions [62], since they don't affect the loop counter. In this respect, Ravi *et al.* [62] proposed to compute the addition operation in the NTT domain (i.e.) compute $\mathbf{z}$ as $\mathsf{INTT}((\hat{\mathbf{s_1}} \circ \hat{c}) + \hat{\mathbf{y}})$ (i.e.) alternative to the computation of $\mathbf{z}$ in Line 27 in Alg.3. 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 an invalid signature which is rejected by the conditional check on $\|\mathbf{z}\|_\infty$ with a very high probability (Line 29 in Alg.3). We propose to utilize the Dynamic_Loop_*Counter* protection along with the addition in the NTT domain, which is referred to as the Verify_Add countermeasure.

*5.2.5 Targeting NTT (*NTT_Twiddle_Fault*).* Ravi *et al.* [66] proposed to inject faults to zeroize the twiddle constants of specific NTT instances in the signing procedure, to generate faulty signatures, which compromise the secret key. They proposed two variants of attacks. The first attack variant works on the deterministic variant of Dilithium, in the following manner. The attacker obtains a valid signature $\sigma = (\mathbf{z}, \mathbf{h}, \mathbf{c})$ of a message μ, with the constraint that $\mathbf{c}[0] = 0$ (first coefficient of $\mathbf{c}$). Let $\mathbf{z} = \mathbf{s}_1 \cdot \mathbf{c} + \mathbf{y}$. Subsequently, the attacker submits a signing query for the same message, but now injects a fault in the NTT instance of $\mathbf{c}$ (Line 26 in Alg.3). This effectively zeroizes the entire NTT output of $\mathbf{c}$ (i.e.) $\hat{\mathbf{c}}$ (line 26). Thus, the generated faulty signature is nothing but $\mathbf{z}^* = \mathbf{y}$. The difference of $\mathbf{z}$ and $\mathbf{z}^*$ can be used to trivially recover $\mathbf{s}_1$, similar to the Generic_DFA attack. This attack only works on the deterministic variant, and cannot work on the probabilistic variant since it is a DFA-style attack. The attack characteristic is denoted using the tuple: (EMFI, Communicate_DUT_IO, Targeted_Fault, 1, 1).

The authors also proposed a non-DFA style variant of the attack that can work on the probabilistic variant, but when $\mathbf{z}$ is computed as $\mathsf{INTT}((\hat{\mathbf{s_1}} \circ \hat{c}) + \hat{\mathbf{y}})$, similar to the Verify_Add countermeasure. They propose to fault the NTT over $\mathbf{y}$ (Line 22), which zeroizes all except the first coefficient of all the polynomials of $\mathbf{y}$. Thus, the resulting faulty signature component $\mathbf{z}^*$ is nothing but $\mathbf{s}_1 \cdot \mathbf{c}$, except for the first coefficient of every polynomial of $\mathbf{z}^*$. The complete secret key $\mathbf{s}_1$ can be recovered in a single such targeted fault. Moreover, the attacker does not require to communicate with the signing DUT for the attack. Thus, the attack characteristic is denoted using the tuple: (EMFI, Observe_DUT_IO, Targeted_Fault, 1, 1).

Countermeasure: The NTT_Twiddle_Check countermeasure that verifies the sanity of the twiddle factors can be used as a concrete countermeasure against the attack (Refer Sec.4.1.2).

Refer to Tab.3 for a tabulation of all fault-injection attacks on the signing procedure of the Dilithium signature scheme.

5.3 FIA on Verification Procedure

While the aforementioned attacks target the signing procedure, the verification procedure could also serve as a good target for fault injection attacks. One of the main motivations being the forceful acceptance of invalid signatures through faults for any message of the attacker's choice.

*5.3.1 Targeting NTT (*NTT_Twiddle_Fault*).* Ravi *et al.* [66] proposed a fault attack that zeroizes the twiddle constants of the NTT over the challenge polynomial $\mathbf{c}$ in the verification procedure (Line 3 in Alg.4). They also proposed a forgery algorithm, which can be used to enforce successful verification for any message of the attacker's choice, if an attacker can achieve the aforementioned fault. We utilize the following tuple to define the attack characteristic: (EMFI,Communicate_DUT_IO, Targeted_Fault, 1, 1).

Countermeasure: The NTT_Twiddle_Check countermeasure that verifies the sanity of the twiddle factors can be used as a concrete countermeasure against the attack (Refer Sec.4.1.2).

5.3.2 *Skipping Equality Check.*

One of the obvious targets for fault injection is to simply skip the final comparison operation that decides the validity of the received signatures. In particular, bypassing the comparison of the received challenge polynomial **c** with the recomputed challenge polynomial $\bar{\mathbf{c}}$ (Line 6 in Alg.4) ensures successful signature verification. This attack is very similar to the Skip_CT_Compare attack on KEMs, targeting the ciphertext comparison operation in the decapsulation procedure. While the attack has not been practically demonstrated, it is important to fortify the equality check in the verification procedure, to prevent trivial skipping attacks. We refer to this attack using the label Skip_C_*Compare*.

Countermeasure (Our Proposal): We propose to simply utilize a Dynamic_Loop_Counter countermeasure to keep track of the number of compared coefficients of the challenge polynomial. This loop counter information along with the result of the comparison operation can be used to protect against trivial skipping attacks. One can also adopt redundancy of varying degrees to further fortify the verification procedure. We agree that this is only an implementation-level countermeasure and can therefore be circumvented by a more powerful attacker. However, these countermeasures do significantly increase the ability of an attacker to mount a successful attack.

Refer to Tab.3 for a tabulation of all fault-injection attacks on the verification procedure of the Dilithium signature scheme.

Table 3. Tablulation of reported FIA and their characteristics for the different procedures of Dilithium signature scheme

Attack	Attack Characteristic					
	Attack_Vector	DUT_IO_Access	Targeted_Or_Not	Num_Faults	Num_Executions	Countermeasure
Key Generation						
Nonce_Fault [64]	EMFI	Observe_DUT_IO	Targeted_Fault	1 – 10	1	Verify_Nonce_Fault
NTT_Twiddle_Fault [66]	EMFI	Observe_DUT_IO	Targeted_Fault	1	1	NTT_Twiddle_Check
Signing						
Randomize_Secret_Key [12, 36]	EMFI	Observe_DUT_IO	Targeted_Fault	1	$1k - 2k$	Verify_After_Sign
Generic_DFA [15]	Glitching	Communicate_DUT_IO	Non_Targeted_Fault	1	1	Verify_After_Sign
Loop_Abort_Fault [26]	Glitching	Observe_DUT_IO	Targeted_Fault	1	1	Verify_Loop_Abort
Skip_Addition [12, 62]	EMFI	Communicate_DUT_IO	Targeted_Fault	1	$1k - 2k$	Verify_Add
NTT_Twiddle_Fault [66]	EMFI	Communicate_DUT_IO	Targeted_Fault	1	1	NTT_Twiddle_Check
	EMFI	Observe_DUT_IO	Targeted_Fault	1	1	NTT_Twiddle_Check
Verification						
NTT_Twiddle_Fault [66]	EMFI	Communicate_DUT_IO	Targeted_Fault	1	1	NTT_Twiddle_Check
Skip_C_*Compare* [76]	Glitching	Communicate_DUT_IO	Targeted_Fault	1	1	Dynamic_Loop_Counter

6 SIDE-CHANNEL ATTACKS ON DILITHIUM

In this section, we discuss side-channel attacks that are applicable to the Dilithium signature scheme. We only consider side-channel attacks on the key-generation and signing procedure as they manipulate the secret key, while the verification procedure which manipulates public information is not relevant for side-channel attacks. We utilize the same characteristics to describe SCA on Dilithium, that were used to describe SCA on Kyber (Refer Sec.3.1.4). We utilize the algorithm of the Dilithium signature scheme in Alg.3-4 to explain the different attacks.

6.1 SCA on Key Generation

6.1.1 SASCA. The key generation procedure of Dilithium is susceptible to SASCA-based attacks.

Targeting NTT (SASCA_NTT): Leakage from the NTT instance over the primary secret key component $\mathbf{s}_1$ (Line 5 of KeyGen in Alg.3) can be used to recover $\mathbf{s}_1$, in a single trace.

Targeting KECCAK (SASCA_KECCAK): KECCAK is used as a PRNG within the key-generation procedure to sample the secret $\mathbf{s}_1$ (Line 3 in Alg.3) using $seed_S$, thus SASCA on this KECCAK instance can be potentially used to recover $seed_S$, which can be used to reconstruct the secret key $\mathbf{s}_1$.

The SASCA_NTT and SASCA_KECCAK attacks can be defined using the tuple: (Observe_DUT_IO, Profiled_With_Clone, 1, High_SNR).

Countermeasure: Shuffling the sensitive NTT, as well as the KECCAK operations, provides concrete protection against attacks relying on SASCA.

6.1.2 Simple Template Attacks. Han *et al.* [32] targeted the NTT instance over $\mathbf{s}_1$ using a simple template attack, which could recover the complete secret polynomial $\mathbf{s}_1$ in a single trace. They showed that an attacker can target leakage from the product of the secret coefficients with the twiddle factors in the first round of the NTT (i.e.) $prod \in \mathbb{Z}_q = \mathbf{s}[i] \cdot \omega^j$ where $\mathbf{s}[i]$ is a secret coefficient and ω^j is a twiddle factor. They show that leakage of the result $prod$ can be used to uniquely distinguish every candidate of $\mathbf{s}[i]$ through simple template attacks. Moreover, the attack is aided by the fact that there are only 5 possible candidates for coefficients of the secret $\mathbf{s}_1$. Han *et al.* [32] targeted the reference implementation of Dilithium through the power side-channel on the ARM Cortex-M4 microcontroller to recover the entire secret $\mathbf{s}_1$ in a single trace. We refer to this attack as the Simple_NTT_Template attack. Similar to SASCA on the NTT, the attack can be defined using the tuple: (Observe_DUT_IO, Profiled_With_Clone, 1, High_SNR).

Countermeasure: Unlike SASCA on NTT which relies on leakage from intermediate variables throughout the NTT operation, this attack only relies on leakage from a single intermediate variable for key recovery. Thus, the leakage exploited is more fine-grained and is more prone to noise (horizontal/vertical) compared to SASCA-type attacks. Nevertheless, the shuffling and masking countermeasures proposed for the NTT serve as a concrete countermeasure against this attack (Shuffled_Masked_NTT [63]).

Refer to Tab.4 for a tabulation of all side-channel attacks on the key-generation procedure of the Dilithium signature scheme.

6.2 SCA on Signing Procedure

6.2.1 SASCA. The signing procedure of Dilithium is susceptible to SASCA-based attacks.

Targeting NTT (SASCA_NTT): Similar to the key-generation procedure, the signing procedure also computes NTT of the primary secret $\mathbf{s}_1$ (Line 15 of Sign Alg.3), which can be targeted using SASCA for single trace key recovery. Similarly, NTT instance over the ephemeral nonce $\mathbf{y}$ (Line 23) can also be targeted, whose knowledge can be used to recover the primary secret $\mathbf{s}_1$.

Targeting KECCAK (SASCA_KECCAK): KECCAK is used as a PRNG within the signing procedure to sample the ephemeral nonce $\mathbf{y}$ from a small seed ρ (Line 22), which is vulnerable to single-trace SASCA_KECCAK attacks.

The SASCA_NTT and SASCA_KECCAK attacks can work on both the deterministic and probabilistic variants of Dilithium and can be defined using the tuple: (Observe_DUT_IO, Profiled_With_Clone, 1, High_SNR).

Countermeasure: Shuffling the NTT as well as the KECCAK operation provides concrete protection against attacks relying on SASCA.

6.2.2 Targeting the Nonce $\mathbf{y}$*.* Recently, Marzougui *et al.* [43] demonstrated a profiled attack targeting the sampling of the ephemeral nonce $\mathbf{y}$ (Line 22 in Alg.22). They proposed to profile the leakage of coefficients of $\mathbf{y}$ using a machine learning classifier, to differentiate between a given coefficient $\mathbf{y}[i] = 0$ and $\mathbf{y} \neq 0$. Templates for the coefficients of $\mathbf{y}$ can be built using leakage from a clone device. During the attack phase, a single trace is obtained and the attacker attempts to exploit leakage from single coefficients of $\mathbf{y}$ to identify zero coefficients of $\mathbf{y}$. If a given coefficient $\mathbf{y}[i] = 0$, then $\mathbf{z}[i] = \mathbf{s}_1 \cdot \mathbf{c}[i]$, and if an attacker can identify $\ell \cdot n$ such coefficients, he can fully recover the secret key using simple Gaussian elimination. Marzougui *et al.* [43] performed their attack on ARM Cortex-M4 microcontroller through the power side-channel, and were able to recover the full key in $\approx 750k$ signatures. A very high number of signatures are required to identify coefficients of $\mathbf{y}$ that have a very small value close to 0, while they are uniformly distributed in the range $[0, 2^19]$ for recommended parameters of Dilithium. The attack also exploits fine leakages from the manipulation of single coefficients and therefore requires a relatively high SNR. We refer to this attack as the Zero_Nonce_Detect attack, which can be described using the tuple: (Observe_DUT_IO, Profiled_With_Clone, $\approx 750k$, High_SNR).

Countermeasure: Masking the nonce $\mathbf{y}$ in the signing procedure serves as an effective countermeasure against the Zero_Nonce_Detect attack, as detecting the exact value of a coefficient using leakage from multiple shares in a single trace is not very trivial, or at the least exponentially increases the number of traces with increasing masking order. There have been several proposals for masking Dilithium against side-channel attacks [7, 44]. We refer to this countermeasure using the label Masking.

6.2.3 Correlation Power Analysis (CPA). The first CPA style attack was proposed by Ravi *et al.* [61], who demonstrated a single-trace horizontal style DPA attack targeting the operation $\mathbf{s}_1 \cdot \mathbf{c}$ (Line 27 in Alg.3), implemented using the school-book polynomial multiplier. However, they only demonstrated a simulated attack assuming idealized leakage models, and to some extent, evaluated the effect of leakage noise. Moreover, NTT is the actual polynomial multiplication algorithm used in Dilithium, and thus this attack is no more applicable to the latest implementations of Dilithium.

More recently, Chen *et al.* [16] demonstrated a non-profiled CPA attack targeting the pointwise multiplication of $\hat{\mathbf{c}}$ and $\hat{\mathbf{s}}_1$ in the NTT domain. They were able to recover the secret key in only 200 power traces using leakage from the ARM Cortex-M4 microcontroller. We refer to these attacks using the label NTT_Leakage_CPA. Since these attack work over multiple traces, they can still work with low SNR. The attack characteristic can be described using the following tuple: (Observe_DUT_IO, Non_Profiled, ≈ 200, Low_SNR). More recently, Steffen *et al.* [70] extended the CPA attack to also target the same pointwise multiplication operation in a hardware implementation on the Artix-7 FPGA, where they required about $66k$ traces for full key recovery, which is ≈ 300 times higher compared to targeting a software implementation on the ARM Cortex-M4 microcontroller. The NTT_Leakage_CPA attack can work on both the deterministic and probabilistic variants of Dilithium.

Countermeasure: Masking the signing procedure serves as a strong countermeasure against the aforementioned CPA-style attacks.

Refer to Tab.4 for a tabulation of all side-channel attacks on the signing procedure of the Dilithium signature scheme.

Table 4. Tablulation of reported SCA and their characteristics for the different procedures of Dilithium signature scheme

Attack	Attack Characteristic					
	Attack_Vector	DUT_IO_Access	Profile_*Requirement*	No_Traces	SNR	Countermeasure
Key Generation						
SASCA_NTT [53, 55]	Power/EM	Observe_DUT_IO	Profiled_With_Clone	1	High_SNR	Shuffled_Masked_NTT
SASCA_KECCAK [37]	Power/EM	Observe_DUT_IO	Profiled_With_Clone	1	High_SNR	Shuffled_KECCAK
Simple_NTT_Template [32]	Power/EM	Observe_DUT_IO	Profiled_With_Clone	1	High_SNR	Shuffled_Masked_NTT
Sign						
SASCA_NTT [53, 55]	Power/EM	Observe_DUT_IO	Profiled_With_Clone	1	High_SNR	Shuffled_Masked_NTT
SASCA_KECCAK [37]	Power/EM	Observe_DUT_IO	Profiled_With_Clone	1	High_SNR	Shuffled_KECCAK
Zero_Nonce_Detect [43]	Power/EM	Observe_DUT_IO	Profiled_With_Clone	$\approx 750k$	High_SNR	Masking
NTT_Leakage_CPA [16, 70]	Power/EM	Observe_DUT_IO	Non_Profiled	≈ 200	Low_SNR	Masking

7 EXPERIMENTAL EVALUATION

We can clearly see that a majority of attacks on both Kyber and Dilithium have been performed on the ARM Cortex-M4 microcontroller. Thus, we perform a practical performance evaluation of the dedicated countermeasures for both Kyber and Dilithium on the same platform. In particular, we implement the countermeasures on the optimized implementation of Kyber and Dilithium from the *pqm4* library [38].

7.1 Target Platform and Implementation Details

Our target platform for the ARM Cortex-M4 processor is the STM32F4DISCOVERY board (DUT) housing the STM32F407 microcontroller and the clock frequency is 24 MHz. Our countermeasures were implemented on the M4-optimized implementations of Kyber and Dilithium available in the public *pqm4* library [38], a benchmarking framework for PQC schemes on the ARM Cortex-M4 microcontroller. The M4-optimized implementation of Kyber is based on the memory-efficient high-speed implementation proposed by Botros, Kannwischer, and Schwabe in [14]. The M4-optimized implementation is based on compact Dilithium optimizations reported by Greconici, Kannwischer, and Sprenkels in [28]. [1] Their work builds upon the early evaluation optimization by Ravi et al. [59] and additionally proposes faster assembly implementations of NTT for the Cortex-M4. All implementations were compiled with the arm-none-eabi-gcc-7.3.1 compiler using compiler flags -O3 -mthumb -mcpu=cortex-m4 -mfloat-abi=hard -mfpu=fpv4-sp-d16. We have implemented the countermeasures on both Kyber and Dilithium such that, the required countermeasures can be independently turned on/off based on the designer's security requirements.

7.2 Experimental Results for Kyber KEM

Considering the different SCA and FIA mounted on Kyber, we implement the following countermeasures within the implementation of Kyber KEM.

(1) Shuffled_Masked_NTT (KeyGen, Encaps, Decaps)
(2) Verify_Nonce_Fault (KeyGen, Encaps)
(3) CT_Sanity_Check (Decaps)

[1] Our analysis and experiments were carried out on the implementations of Kyber and Dilithium corresponding to the commit hash 2691b4915b76db8b765ba89e4e09adc6b999763f, and were available in the *pqm4* library until Jan 31, 2022.

(4) Message_Poly_Sanity_Check (Decaps)
(5) Protect_CT_Compare (Decaps)
(6) Shuffled_Encode and Shuffled_Encode (Encaps, Decaps)

While we have also discussed dedicated countermeasures such as Shuffled_KECCAK and NTT_Twiddle_Check in the paper, we have not implemented them for Kyber KEM in this work. Nevertheless, the aforementioned dedicated countermeasures either separate or combined, are not meant to serve as standalone countermeasures for Kyber but can be implemented on top of masking countermeasures for concrete protection against both SCA and FIA [13, 33].

Refer to Tab.5 for the performance overheads due to the Shuffled_Masked_NTT countermeasures against the NTT_Leakage attacks on Kyber KEM, running on the ARM Cortex-M4 microcontroller. While we have implemented all the shuffling (3) and masking (4) countermeasures proposed in [63], for brevity, we only report numbers for the countermeasures referred to as Coarse_Shuffled_NTT and Generic_2_Masked_NTT (Refer to [63] for the terminology used for the different Shuffled_Masked_NTT countermeasures).

On the ARM Cortex-M4 device, we observe a performance impact in the range of $52-69\%$ for key generation, 44-74% for the encapsulation, and $52-96\%$ for the decapsulation procedure, across all parameters of Kyber KEM. Please note that the unprotected implementation utilizes assembly-optimized NTT, while the protected implementation utilizes protected NTTs which are implemented in C. Thus, we argue that it is possible to obtain significantly improved overheads, provided that the protected NTT/INTTs are implemented in assembly. We leave the optimized implementation of these countermeasures in assembly as future work.

Table 5. Performance Comparison of the Shuffled_Masked_NTT countermeasures for Kyber KEM, compared to the optimized unprotected implementations on the ARM Cortex-M4 device. Numbers were obtained on the STM32F407VG microcontroller mounted on the STM32F407DISCOVERY board, running at 24 MHz. Numbers are provided in terms of $\times10^3$ clock cycles. Ovh denotes overhead in percentage.

Scheme	**Clock Cycles ($\times10^3$)**								
	KeyGen			**Encaps**			**Decaps**		
	Unprot.	**Prot.**	**Ovh. (%)**	**Unprot.**	**Prot.**	**Ovh. (%)**	**Unprot.**	**Prot.**	**Ovh. (%)**
Coarse_Shuffled_NTT									
Kyber512	463	786	69	556	971	74	518	1021	96
Kyber768	762	1245	63	909	1486	63	853	1512	77
Kyber1024	1207	1854	53	1386	2125	53	1312	2133	62
Generic_2_Masked_NTT									
Kyber512	463	732	57	556	899	61	518	937	80
Kyber768	761	1163	52	909	1387	52	853	1406	64
Kyber1024	1207	1744	44	1386	1998	44	1312	1999	52

Refer to Tab.6 for the performance overheads due to the Verify_Nonce_Fault countermeasure on the key-generation procedure, and CT_Sanity_Check, Message_Poly_Sanity_Check, Shuffle_Encode and Shuffled_Decode countermeasures for the decapsulation procedure for Kyber KEM, implemented on the ARM Cortex-M4 device. These countermeasures impose very reasonable overheads in the range of 10-11%, 15-34%, 12-30%, and 4-5% for the different parameter

Table 6. Performance Comparison of the custom SCA-FIA countermeasures for Kyber KEM, compared to the optimized unprotected implementation on the ARM Cortex-M4 device. Numbers were obtained on the STM32F407VG microcontroller mounted on the STM32F407DISCOVERY board, running at 24 MHz. Numbers are provided in terms of $\times 10^3$ clock cycles. Ovh denotes overhead in percentage.

Scheme	Clock Cycles ($\times 10^3$)		
	Unprot.	**Prot.**	**Ovh. (%)**
Verify_Nonce_Fault (KeyGen)			
Kyber512	463	516	11
Kyber768	762	848	11
Kyber1024	1207	1337	10
CT_Sanity_Check (Decaps)			
Kyber512	518	698	34
Kyber768	853	1040	21
Kyber1024	1312	1520	15
Message_Poly_Sanity_Check (Decaps)			
Kyber512	518	679	30
Kyber768	853	1014	18
Kyber1024	1312	1473	12
Protect_CT_Compare (Decaps)			
Kyber512	518	549	5
Kyber768	853	894	4
Kyber1024	1312	1372	4
Shuffle_Encode_Decode (Decaps)			
Kyber512	518	586	13
Kyber768	853	878	2
Kyber1024	1312	1337	2

sets of Kyber KEM. Thus, we can see that these dedicated countermeasures can be implemented in a cost-effective manner for Kyber KEM.

7.3 Experimental Results for Dilithium

Considering the different SCA and FIA mounted on Dilithium, we implement the following countermeasures within the implementation of the Dilithium signature scheme.

(1) Shuffled_Masked_NTT (KeyGen, Sign)
(2) Verify_After_Sign (Sign)
(3) Verify_Loop_Abort (Sign)
(4) Verify_Add (Sign)
(5) Protect_Verify_Compare (Verify)

While we have also discussed dedicated countermeasures such as Shuffled_KECCAK and NTT_Twiddle_Check and Verify_Nonce_Fault countermeasures in the paper, we have not implemented them for Dilithium in this work. Nevertheless, the aforementioned dedicated countermeasures either separate or combined, are not meant to serve as standalone countermeasures for Dilithium, but can be implemented on top of masking countermeasures for concrete protection against both SCA and FIA [7, 44].

Refer to Tab.7 for the performance overheads due to the Shuffled_Masked_NTT countermeasures against the NTT_Leakage attacks on Dilithium implemented on the ARM Cortex-M4 microcontroller. While we have implemented all the shuffling (3) and masking (4) countermeasures proposed in [63], for brevity, we only report numbers for the countermeasures referred to as Coarse_Shuffled_NTT and Generic_2_Masked_NTT.

On the ARM Cortex-M4 device, we observe a performance impact in the range of 22 − 32% for key generation and 116 − 132% for the signing procedure. Please note that the unprotected implementation utilizes assembly-optimized NTT, while the protected implementation utilizes protected NTTs which are implemented in C. The overhead on the signing procedure is much more pronounced since the majority of its computation time is consumed by the polynomial multiplication operation. Moreover, the iterative nature of the signing procedure further increases the impact of our unoptimized protected NTT implementations. Thus, we argue that it is possible to obtain significantly improved overheads, provided that the protected NTT/INTTs are implemented in assembly. We leave the optimized implementation of these countermeasures in assembly as future work.

Table 7. Performance Comparison of the Shuffled_Masked_NTT countermeasures for Dilithium, compared to the optimized unprotected implementation on the ARM Cortex-M4 device. Numbers were obtained on the STM32F407VG microcontroller mounted on the STM32F407DISCOVERY board, running at 24 MHz. Numbers are provided in terms of $\times 10^6$ clock cycles. Ovh denotes overhead in percentage.

Scheme	**Clock Cycles** ($\times 10^6$)					
	KeyGen			**Sign**		
	Unprot.	**Prot.**	**Ovh. (%)**	**Unprot.**	**Prot.**	**Ovh. (%)**
Coarse_Shuffled_NTT						
Dilithium2	1.6	2.1	32	4.1	9.2	124
Dilithium3	2.8	3.5	24	6.6	15.3	132
Generic_2_Masked_NTT						
Dilithium2	1.6	2.0	30	4.1	8.9	116
Dilithium3	2.8	3.5	22	6.6	14.6	121

Refer to Tab.8 for the performance overheads due to the Verify_After_Sign, Verify_Loop_Abort and Verify_Add countermeasures for the signing procedure and Protect_Verify_Compare countermeasure for the verification procedure of Dilithium, implemented on the ARM Cortex-M4 microcontroller. On the ARM Cortex-M4 device, these countermeasures impose very reasonable overheads in the range of 8-12%, 11-13%, and 1 − 2% and 0.1 − 0.05% for the different parameter sets of Dilithium. Thus, we can see that these dedicated countermeasures can be implemented in a cost-effective manner for the Dilithium signature scheme.

8 CONCLUSION

In this work, we present a systematic study of Side-Channel Attacks (SCA) and Fault Injection Attacks (FIA) on structured lattice-based schemes, with a focus on Kyber and Dilithium, and also discuss appropriate countermeasures for each of the different attacks. Among the several countermeasures discussed in this work, we present novel countermeasures

Table 8. Performance Comparison of the different SCA-FIA countermeasures for Dilithium and the overheads they incur on on optimized implementations on the ARM Cortex-M4 device. Numbers were obtained on the STM32F407VG microcontroller mounted on the STM32F407DISCOVERY board, running at 24 MHz. Numbers are provided in terms of $\times 10^6$ clock cycles. Ovh denotes overhead in percentage.

Scheme	**Clock Cycles ($\times 10^6$)**		
	Unprot.	**Prot.**	**Ovh. (%)**
Verify_After_Sign (Sign)			
Dilithium2	4.1	4.6	12
Dilithium3	6.6	7.2	8
Verify_Loop_Abort (Sign)			
Dilithium2	4.1	4.7	13
Dilithium2	6.6	7.3	11
Verify_Add (Sign)			
Dilithium2	4.1	4.3	2
Dilithium2	6.6	6.7	1
Protect_Verify_Compare (Verify)			
Dilithium2	1.5	1.5	≈ 0.10
Dilithium3	2.6	2.7	≈ 0.05

that offer simultaneous protection against several SCA and FIA-based chosen-ciphertext attacks for Kyber KEM. We implement the presented countermeasures within the well-known *pqm4* library for the ARM Cortex-M4 based microcontroller, which incurs reasonable performance overheads on the target platform. We therefore believe our work argues for the usage of custom countermeasures within real-world implementations of lattice-based schemes, either in a standalone manner or as reinforcements to generic countermeasures such as masking.

ACKNOWLEDGEMENT

We would like to thank Dr. Sujoy Sinha Roy from TU Graz for very useful discussions regarding categorization of side-channel and fault attacks on lattice-based schemes.

This work was supported in part by CyberSecurity Research Flanders with reference number VR20192203, the Research Council KU Leuven (C16/15/058) and the Horizon 2020 ERC Advanced Grant (101020005 Belfort). Jan-Pieter D'Anvers is funded by FWO (Research Foundation – Flanders) as junior post-doctoral fellow (contract number 133185 / 1238822N LV).

REFERENCES

[1] 2016. The transport layer security (TLS) protocol version 1.3 (May 2016). https://tools.ietf.org/html/draft-ietf-tls-tls13-13.

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

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

[4] Dorian Amiet, Andreas Curiger, Lukas Leuenberger, and Paul Zbinden. 2020. Defeating NewHope with a single trace. In *International Conference on Post-Quantum Cryptography*. Springer, 189–205.

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

[6] Roberto Avanzi, Joppe W. Bos, Leo Ducas, Eike Kiltz, Tancrede Lepoint, Vadim Lyubashevsky, John Schanck, Peter Schwabe, Gregor Seiler, and Damien Stehlé. 2021. CRYSTALS-Kyber (version 3.02): Algorithm specifications and supporting documentation (Auguest 4, 2021). https://pq-crystals.org/kyber/data/kyber-specification-round3.pdf. (2021).

[7] Melissa Azouaoui, Olivier Bronchain, Gaëtan Cassiers, Clément Hoffmann, Yulia Kuzovkova, Joost Renes, Markus Schönauer, Tobias Schneider, François-Xavier Standaert, and Christine van Vredendaal. 2022. Leveling Dilithium against Leakage: Revisited Sensitivity Analysis and Improved Implementations. *Cryptology ePrint Archive* (2022).

[8] Florian Bache, Clara Paglialonga, Tobias Oder, Tobias Schneider, and Tim Güneysu. 2020. High-Speed Masking for Polynomial Comparison in Lattice-based KEMs. *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2020, 3 (2020), 483–507. https://doi.org/10.13154/tches.v2020.i3.483-507

[9] Ciprian Baetu, F. Betül Durak, Loïs Huguenin-Dumittan, Abdullah Talayhan, and Serge Vaudenay. 2019. Misuse Attacks on Post-quantum Cryptosystems. In *Advances in Cryptology - EUROCRYPT 2019 - 38th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Darmstadt, Germany, May 19-23, 2019, Proceedings, Part II (Lecture Notes in Computer Science, Vol. 11477)*, Yuval Ishai and Vincent Rijmen (Eds.). Springer, 747–776. https://doi.org/10.1007/978-3-030-17656-3_26

[10] Michiel Van Beirendonck, Jan-Pieter D'anvers, Angshuman Karmakar, Josep Balasch, and Ingrid Verbauwhede. 2021. A side-channel-resistant implementation of SABER. *ACM Journal on Emerging Technologies in Computing Systems (JETC)* 17, 2 (2021), 1–26.

[11] Shivam Bhasin, Jan-Pieter D'Anvers, Daniel Heinz, Thomas Pöppelmann, and Michiel van Beirendonck. 2021. Attacking and Defending Masked Polynomial Comparison for Lattice-Based Cryptography. 2021, 3 (2021), 334–359. https://doi.org/10.46586/tches.v2021.i3.334-359

[12] Nina Bindel, Johannes Buchmann, and Juliane Krämer. 2016. 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*. IEEE Computer Society, 63–77. https://doi.org/10.1109/FDTC.2016.11

[13] Joppe W. Bos, Marc Gourjon, Joost Renes, Tobias Schneider, and Christine van Vredendaal. 2021. Masking Kyber: First- and Higher-Order Implementations. *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2021, 4 (2021), 173–214. https://doi.org/10.46586/tches.v2021.i4.173-214

[14] Leon Botros, Matthias J. Kannwischer, and Peter Schwabe. 2019. Memory-Efficient High-Speed Implementation of Kyber on Cortex-M4. In *Progress in Cryptology - AFRICACRYPT 2019 - 11th International Conference on Cryptology in Africa, Rabat, Morocco, July 9-11, 2019, Proceedings* (2019). 209–228. https://doi.org/10.1007/978-3-030-23696-0_11

[15] Leon Groot Bruinderink and Peter Pessl. 2018. 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.

[16] Zhaohui Chen, Emre Karabulut, Aydin Aysu, Yuan Ma, and Jiwu Jing. 2021. An Efficient Non-Profiled Side-Channel Attack on the CRYSTALS-Dilithium Post-Quantum Signature. In *39th IEEE International Conference on Computer Design, ICCD 2021, Storrs, CT, USA, October 24-27, 2021*. IEEE, 583–590. https://doi.org/10.1109/ICCD53106.2021.00094

[17] Eric Crockett, Christian Paquin, and Douglas Stebila. 2019. Prototyping post-quantum and hybrid key exchange and authentication in TLS and SSH. https://github.com/open-quantum-safe/openssl. *IACR Cryptol. ePrint Arch.* (2019), 858. https://eprint.iacr.org/2019/858

[18] Dana Dachman-Soled, Léo Ducas, Huijing Gong, and Mélissa Rossi. 2020. LWE with Side Information: Attacks and Concrete Security Estimation. In *Advances in Cryptology – CRYPTO 2020*, Daniele Micciancio and Thomas Ristenpart (Eds.). Springer International Publishing, Cham, 329–358.

[19] Jan-Pieter D'Anvers, Daniel Heinz, Peter Pessl, Michiel Van Beirendonck, and Ingrid Verbauwhede. 2022. Higher-Order Masked Ciphertext Comparison for Lattice-Based Cryptography. *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2022, 2 (2022), 115–139. https://doi.org/10.46586/tches.v2022.i2.115-139

[20] Jan-Pieter D'Anvers, Qian Guo, Thomas Johansson, Alexander Nilsson, Frederik Vercauteren, and Ingrid Verbauwhede. 2019. Decryption Failure Attacks on IND-CCA Secure Lattice-Based Schemes. In *Public-Key Cryptography – PKC 2019*, Dongdai Lin and Kazue Sako (Eds.). Springer International Publishing, Cham, 565–598.

[21] Jan-Pieter D'Anvers, Mélissa Rossi, and Fernando Virdia. 2020. (One) Failure Is Not an Option: Bootstrapping the Search for Failures in Lattice-Based Encryption Schemes. In *Advances in Cryptology – EUROCRYPT 2020*, Anne Canteaut and Yuval Ishai (Eds.). Springer International Publishing, Cham, 3–33.

[22] Jan-Pieter D'Anvers, Marcel Tiepelt, Frederik Vercauteren, and Ingrid Verbauwhede. 2019. Timing attacks on error correcting codes in post-quantum schemes. In *Proceedings of ACM Workshop on Theory of Implementation Security Workshop*. 2–9.

[23] Jeroen Delvaux. 2021. Roulette: Breaking Kyber with Diverse Fault Injection Setups. *Cryptology ePrint Archive* (2021), 1622.

[24] Léo Ducas, Alain Durmus, Tancrède Lepoint, and Vadim Lyubashevsky. 2013. Lattice Signatures and Bimodal Gaussians. In *Advances in Cryptology - CRYPTO 2013 - 33rd Annual Cryptology Conference, Santa Barbara, CA, USA, August 18-22, 2013. Proceedings, Part I (Lecture Notes in Computer Science, Vol. 8042)*, Ran Canetti and Juan A. Garay (Eds.). Springer, 40–56. https://doi.org/10.1007/978-3-642-40041-4_3

[25] Léo Ducas, Tancrede Lepoint, Vadim Lyubashevsky, Peter Schwabe, Gregor Seiler, and Damien Stehlé. 2018. Crystals–dilithium: Digital signatures from module lattices. https://pq-crystals.org/dilithium/data/dilithium-specification-round3.pdf. *Submission to the NIST's post-quantum cryptography standardization process* (2018).

[26] Thomas Espitau, Pierre-Alain Fouque, Benoît Gérard, and Mehdi Tibouchi. 2016. Loop-abort faults on lattice-based fiat-shamir and hash-and-sign signatures. In *International Conference on Selected Areas in Cryptography*. Springer, 140–158.

[27] Eiichiro Fujisaki and Tatsuaki Okamoto. 1999. Secure integration of asymmetric and symmetric encryption schemes. In *Annual international cryptology conference*. Springer, 537–554.

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

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

[30] Qian Guo, Thomas Johansson, and Alexander Nilsson. 2020. A key-recovery timing attack on post-quantum primitives using the Fujisaki-Okamoto transformation and its application on FrodoKEM. In *Annual International Cryptology Conference*. Springer, 359–386.

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

[32] Jaeseung Han, Taeho Lee, Jihoon Kwon, Joohee Lee, Il-Ju Kim, Jihoon Cho, Dong-Guk Han, and Bo-Yeon Sim. 2021. Single-Trace Attack on NIST Round 3 Candidate Dilithium Using Machine Learning-Based Profiling. *IEEE Access* 9 (2021), 166283–166292. https://doi.org/10.1109/ACCESS.2021.3135600

[33] Daniel Heinz, Matthias J. Kannwischer, Georg Land, Thomas Pöppelmann, Peter Schwabe, and Daan Sprenkels. 2022. First-Order Masked Kyber on ARM Cortex-M4. *IACR Cryptol. ePrint Arch.* (2022), 58. https://eprint.iacr.org/2022/058

[34] Julius Hermelink, Peter Pessl, and Thomas Pöppelmann. 2021. Fault-enabled chosen-ciphertext attacks on kyber. In *International Conference on Cryptology in India*. Springer, 311–334.

[35] Julius Hermelink, Silvan Streit, Emanuele Strieder, and Katharina Thieme. 2022. Adapting Belief Propagation to Counter Shuffling of NTTs. *IACR Cryptol. ePrint Arch.* (2022), 555. https://eprint.iacr.org/2022/555

[36] Saad Islam, Koksal Mus, Richa Singh, Patrick Schaumont, and Berk Sunar. 2022. Signature Correction Attack on Dilithium Signature Scheme. In *7th IEEE European Symposium on Security and Privacy, EuroS&P 2022, Genoa, Italy, June 6-10, 2022*. IEEE, 647–663. https://doi.org/10.1109/EuroSP53844.2022.00046

[37] Matthias J. Kannwischer, Peter Pessl, and Robert Primas. 2020. Single-Trace Attacks on Keccak. *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2020, 3 (2020), 243–268. https://doi.org/10.13154/tches.v2020.i3.243-268

[38] Matthias J. Kannwischer, Joost Rijneveld, Peter Schwabe, and Ko Stoffelen. 2019. PQM4: Post-quantum crypto library for the ARM Cortex-M4. https://github.com/mupq/pqm4.

[39] Yanbin Li, Jiajie Zhu, Yuxin Huang, Zhe Liu, and Ming Tang. 2022. Single-Trace Side-Channel Attacks on the Toom-Cook: The Case Study of Saber. *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2022, 4 (2022), 285–310. https://doi.org/10.46586/tches.v2022.i4.285-310

[40] Xianhui Lu, Yamin Liu, Zhenfei Zhang, Dingding Jia, Haiyang Xue, Jingnan He, and Bao Li. 2018. LAC: Practical Ring-LWE Based Public-Key Encryption with Byte-Level Modulus. *IACR Cryptol. ePrint Arch.* (2018), 1009. https://eprint.iacr.org/2018/1009

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

[42] Vadim Lyubashevsky, Chris Peikert, and Oded Regev. 2013. On Ideal Lattices and Learning with Errors over Rings. *J. ACM* 60, 6 (2013), 43.

[43] Soundes Marzougui, Vincent Ulitzsch, Mehdi Tibouchi, and Jean-Pierre Seifert. 2022. Profiling Side-Channel Attacks on Dilithium: A Small Bit-Fiddling Leak Breaks It All. *IACR Cryptol. ePrint Arch.* (2022), 106. https://eprint.iacr.org/2022/106

[44] Vincent Migliore, Benoît Gérard, Mehdi Tibouchi, and Pierre-Alain Fouque. 2019. Masking Dilithium - Efficient Implementation and Side-Channel Evaluation. In *Applied Cryptography and Network Security - 17th International Conference, ACNS 2019, Bogota, Colombia, June 5-7, 2019, Proceedings (Lecture Notes in Computer Science, Vol. 11464)*, Robert H. Deng, Valérie Gauthier-Umaña, Martín Ochoa, and Moti Yung (Eds.). Springer, 344–362. https://doi.org/10.1007/978-3-030-21568-2_17

[45] Catinca Mujdei, Arthur Beckers, Jose Bermundo, Angshuman Karmakar, Lennert Wouters, and Ingrid Verbauwhede. 2022. Side-Channel Analysis of Lattice-Based Post-Quantum Cryptography: Exploiting Polynomial Multiplication. *IACR Cryptol. ePrint Arch.* (2022), 474. https://eprint.iacr.org/2022/474

[46] Hamid Nejatollahi, Nikil Dutt, Sandip Ray, Francesco Regazzoni, Indranil Banerjee, and Rosario Cammarota. 2019. Post-quantum lattice-based cryptography implementations: A survey. *ACM Computing Surveys (CSUR)* 51, 6 (2019), 1–41.

[47] Kalle Ngo, Elena Dubrova, Qian Guo, and Thomas Johansson. 2021. A side-channel attack on a masked IND-CCA secure Saber KEM implementation. *IACR Transactions on Cryptographic Hardware and Embedded Systems* (2021), 676–707.

[48] Kalle Ngo, Elena Dubrova, and Thomas Johansson. 2021. Breaking Masked and Shuffled CCA Secure Saber KEM by Power Analysis. In *Proceedings of the 5th Workshop on Attacks and Solutions in Hardware Security*. 51–61.

[49] Kalle Ngo, Ruize Wang, Elena Dubrova, and Nils Paulsrud. 2022. Side-Channel Attacks on Lattice-Based KEMs Are Not Prevented by Higher-Order Masking. *IACR Cryptol. ePrint Arch.* (2022), 919. https://eprint.iacr.org/2022/919

[50] Tobias Oder, Tobias Schneider, Thomas Pöppelmann, and Tim Güneysu. 2018. Practical CCA2-Secure and Masked Ring-LWE Implementation. *IACR Transactions on Cryptographic Hardware and Embedded Systems* (2018), 142–174.

[51] Tobias Oder, Tobias Schneider, Thomas Pöppelmann, and Tim Güneysu. 2018. Practical CCA2-Secure and Masked Ring-LWE Implementation. *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2018, 1 (2018), 142–174. https://doi.org/10.13154/tches.v2018.i1.142-174

[52] Judea Pearl. 1986. Fusion, propagation, and structuring in belief networks. *Artificial intelligence* 29, 3 (1986), 241–288.

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

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

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

[56] Yue Qin, Chi Cheng, Xiaohan Zhang, Yanbin Pan, Lei Hu, and Jintai Ding. 2021. A Systematic Approach and Analysis of Key Mismatch Attacks on Lattice-Based NIST Candidate KEMs. In *Advances in Cryptology - ASIACRYPT 2021 - 27th International Conference on the Theory and Application of Cryptology and Information Security, Singapore, December 6-10, 2021, Proceedings, Part IV (Lecture Notes in Computer Science, Vol. 13093)*, Mehdi Tibouchi and Huaxiong Wang (Eds.). Springer, 92–121. https://doi.org/10.1007/978-3-030-92068-5_4

[57] Gokulnath Rajendran, Prasanna Ravi, Jan-Pieter D'Anvers, Shivam Bhasin, and Anupam Chattopadhyay. 2022. Pushing the Limits of Generic Side-Channel Attacks on LWE-based KEMs - Parallel PC Oracle Attacks on Kyber KEM and Beyond. *IACR Cryptol. ePrint Arch.* (2022), 931. https://eprint.iacr.org/2022/931

[58] Prasanna Ravi, Shivam Bhasin, Sujoy Sinha Roy, and Anupam Chattopadhyay. 2021. On Exploiting Message Leakage in (few) NIST PQC Candidates for Practical Message Recovery Attacks. *IEEE Transactions on Information Forensics and Security* (2021).

[59] Prasanna Ravi, Sourav Sen Gupta, Anupam Chattopadhyay, and Shivam Bhasin. 2019. Improving speed of Dilithium's signing procedure. In *International Conference on Smart Card Research and Advanced Applications*. Springer, 57–73.

[60] Prasanna Ravi, James Howe, Anupam Chattopadhyay, and Shivam Bhasin. 2021. Lattice-based key-sharing schemes: A survey. *ACM Computing Surveys (CSUR)* 54, 1 (2021), 1–39.

[61] Prasanna Ravi, Mahabir Prasad Jhanwar, James Howe, Anupam Chattopadhyay, and Shivam Bhasin. 2018. Side-channel assisted existential forgery attack on Dilithium-a NIST PQC candidate. *Cryptology ePrint Archive* (2018).

[62] Prasanna Ravi, Mahabir Prasad Jhanwar, James Howe, Anupam Chattopadhyay, and Shivam Bhasin. 2019. 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*. 427–440.

[63] Prasanna Ravi, Romain Poussier, Shivam Bhasin, and Anupam Chattopadhyay. 2020. On Configurable SCA Countermeasures Against Single Trace Attacks for the NTT. In *International Conference on Security, Privacy, and Applied Cryptography Engineering*. Springer, 123–146.

[64] Prasanna Ravi, Debapriya Basu Roy, Shivam Bhasin, Anupam Chattopadhyay, and Debdeep Mukhopadhyay. 2019. Number "Not Used" Once-Practical Fault Attack on pqm4 Implementations of NIST Candidates. In *International Workshop on Constructive Side-Channel Analysis and Secure Design*. Springer, 232–250.

[65] Prasanna Ravi, Sujoy Sinha Roy, Anupam Chattopadhyay, and Shivam Bhasin. 2020. Generic Side-channel attacks on CCA-secure lattice-based PKE and KEMs. *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2020, 3 (2020), 307–335.

[66] Prasanna Ravi, Bolin Yang, Shivam Bhasin, Fan Zhang, and Anupam Chattopadhyay. 2022. Fiddling the Twiddle Constants - Fault Injection Analysis of the Number Theoretic Transform. *IACR Cryptol. ePrint Arch.* (2022), 824. https://eprint.iacr.org/2022/824

[67] Oded Regev. 2009. On lattices, learning with errors, random linear codes, and cryptography. *Journal of the ACM (JACM)* 56, 6 (2009), 1–40.

[68] Muyan Shen, Chi Cheng, Xiaohan Zhang, Qian Guo, and Tao Jiang. 2022. Find the Bad Apples: An efficient method for perfect key recovery under imperfect SCA oracles â€" A case study of Kyber. *IACR Cryptol. ePrint Arch.* (2022), 563. https://eprint.iacr.org/2022/563

[69] Bo-Yeon Sim, Jihoon Kwon, Joohee Lee, Il-Ju Kim, Tae-Ho Lee, Jaeseung Han, Hyojin Yoon, Jihoon Cho, and Dong-Guk Han. 2020. Single-Trace Attacks on Message Encoding in Lattice-Based KEMs. 8 (2020), 183175–183191.

[70] Hauke Steffen, Georg Land, Lucie Kogelheide, and Tim Güneysu. 2022. Breaking and Protecting the Crystal: Side-Channel Analysis of Dilithium in Hardware. *Cryptology ePrint Archive* (2022).

[71] Yutaro Tanaka, Rei Ueno, Keita Xagawa, Akira Ito, Junko Takahashi, and Naofumi Homma. 2022. Multiple-Valued Plaintext-Checking Side-Channel Attacks on Post-Quantum KEMs. *IACR Cryptol. ePrint Arch.* (2022), 940. https://eprint.iacr.org/2022/940

[72] Rei Ueno, Keita Xagawa, Yutaro Tanaka, Akira Ito, Junko Takahashi, and Naofumi Homma. 2022. Curse of Re-encryption: A Generic Power/EM Analysis on Post-Quantum KEMs. *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2022, 1 (2022), 296–322. https://doi.org/10.46586/tches.v2022.i1.296-322

[73] Nicolas Veyrat-Charvillon, Benoît Gérard, and François-Xavier Standaert. 2014. Soft Analytical Side-Channel Attacks. In *Advances in Cryptology - ASIACRYPT 2014 - 20th International Conference on the Theory and Application of Cryptology and Information Security, Kaoshiung, Taiwan, R.O.C., December 7-11, 2014. Proceedings, Part I (Lecture Notes in Computer Science, Vol. 8873)*, Palash Sarkar and Tetsu Iwata (Eds.). Springer, 282–296. https://doi.org/10.1007/978-3-662-45611-8_15

[74] Ruize Wang, Kalle Ngo, and Elena Dubrova. 2022. Making Biased DL Models Work: Message and Key Recovery Attacks on Saber Using Amplitude-Modulated EM Emanations. *IACR Cryptol. ePrint Arch.* (2022), 852. https://eprint.iacr.org/2022/852

[75] Ruize Wang, Kalle Ngo, and Elena Dubrova. 2022. Side-Channel Analysis of Saber KEM Using Amplitude-Modulated EM Emanations. *IACR Cryptol. ePrint Arch.* (2022), 807. https://eprint.iacr.org/2022/807

[76] Keita Xagawa, Akira Ito, Rei Ueno, Junko Takahashi, and Naofumi Homma. 2021. 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*. Springer, 33–61.

[77] Zhuang Xu, Owen Michael Pemberton, Sujoy Sinha Roy, David Oswald, Wang Yao, and Zhiming Zheng. 2021. Magnifying side-channel leakage of lattice-based cryptosystems with chosen ciphertexts: The case study of kyber. *IEEE Trans. Comput.* (2021).