Beware of Keccak: Practical Fault Attacks on SHA-3 to Compromise Kyber and Dilithium on ARM Cortex-M Devices

Yuxuan Wang, Jintong Yu, Shipei Qu, Xiaolin Zhang, Xiaowei Li, Chi Zhang and Dawu Gu

Shanghai Jiao Tong University, Shanghai, China, 18588297218@sjtu.edu.cn

Abstract. Keccak acts as the hash algorithm and eXtendable-Output Function (XOF) specified in the NIST standard drafts for Kyber and Dilithium. The Keccak output is highly correlated with sensitive information. While in RSA and ECDSA, hash-like components are only used to process public information, such as the message. The importance and sensitivity of hash-like components like Keccak are much higher in Kyber and Dilithium than in traditional public-key cryptography. However, few works study Keccak regarding the physical security of Kyber and Dilithium. In this paper, we propose a practical fault attack scheme on Keccak to compromise Kyber and Dilithium on ARM Cortex-M devices. Specifically, by injecting loop-abort faults in the iterative assignments or updates of Keccak, we propose six attacks that can set the Keccak output to a known value. These attacks can be exploited to disrupt the random number expansion or other critical processes in Kyber and Dilithium, thereby recovering sensitive information derived from the Keccak output. In this way, we propose eight attack strategies on Kyber and seven on Dilithium, achieving key recovery, signature forgery, and verification bypass. To validate the practicality of the proposed attack strategies, we perform fault characterization on five real-world devices belonging to four different series (ARM Cortex-M0+, M3, M4, and M33). The success rate is up to 89.5%, demonstrating the feasibility of loop-abort faults. This paper also provides a guide for reliably inducing loop-abort faults on ARM Cortex-M devices using electromagnetic fault injection. We further validate our complete attacks on Kyber and Dilithium based on the official implementations, achieving a success rate of up to 55.1%. The results demonstrate that the excessive use of Keccak in generating and computing secret information leads to severe vulnerabilities. Our work can potentially be migrated to other lattice-based post-quantum cryptographic algorithms that use Keccak, such as Falcon, BIKE, and HQC.

Keywords: Post-Quantum Cryptography · Fault Injection Attack · Keccak · Kyber · Dilithium · ARM Cortex-M

1 Introduction

The NIST Post-Quantum Cryptography (PQC) standardization has selected two winners, CRYSTALS-Kyber [BDK+18] and CRYSTALS-Dilithium [DKL+18], which will be widely used in Internet of Things (IoT), mobile computing, cloud computing and other real-world applications. For example, Google started supporting Kyber768 in Chrome 116 [chr]. However, their security guarantees can be undermined in practice. Fault Injection Attack (FIA) is a type of invasive physical attack that induces faults during the execution of a cryptographic algorithm. It can cause unexpected outputs that reveal the private keys. Therefore, resistance against FIA is an essential criterion in NIST standardization and has attracted the attention of many researchers.

Licensed under Creative Commons License CC-BY 4.0.

The existing FIAs on Kyber and Dilithium can be roughly divided into four types.

- **Type 1:** Generating weak LWE instances to expose the private keys. Espitau et al. [EFGT18, EFGT17] reduce the entropy of $\mathbf{e}$ by setting some polynomial coefficients to zero, thereby exposing secret $\mathbf{s}$ protected by $\mathbf{e}$. Ulitzsch et al. [UMB+23] improved upon this work, making it applicable to implementations with fault Countermeasures. Ravi et al. [RRB+19] proposed a fault attack targeting the sampling of secrets and errors, creating LWE instances of the form $\mathbf{t} = \mathbf{A} \cdot \mathbf{s} + \mathbf{s}$, but it requires multiple faults in one execution.

- **Type 2:** Skipping critical computations or operations. Ravi et al. [RJH+19] skip the addition $\mathbf{z} = \mathbf{y} + c \cdot \mathbf{s_1}$ during Dilithium signing and generate a signature $\mathbf{z} = c \cdot \mathbf{s_1}$ exposing private key $\mathbf{s_1}$, requiring $1k$-$2k$ attempts. Xagawa et al. [XIU+21] skip the comparative checks of the Kyber decapsulation and the Dilithium verification, exposing $\mathbf{s}$ using a method similar to [DTVV19, RRCB20] or bypassing verification.

- **Type 3:** Ineffective Fault Analysis (IFA) and Fault Correction Attacks (FCA). For Kyber, Prokop and Pessl [PP21] inject faults into the message decoding process during the decapsulation to obtain whether the total error d is greater than 0. Then, they can recover the private key $\mathbf{s}$ using the linear relationship between d and $\mathbf{s}$ within $6.5k$-$13k$ fault injections. Hermelink et al. [HPP21] improved this method, reducing the number of fault injections to $5k$-$7k$. Delvaux et al. [DDP21] significantly increased the time window of the attack but requires more than $10k$ fault injections. For Dilithium, Islam et al. [IMS+22] use Rowhammer [KDK+14] to induce bitflips in the private keys, and then use a FCA to recover the keys from $1k$-$2k$ faulty signatures. Recently, Krahmer et al. [KPLG24] recovered the private key by injecting faults into matrix $\mathbf{A}$ within $0.5k$-$1k$ injections.

- **Type 4:** Differential Fault Analysis (DFA). Bruinderink and Pessl [BP18] generate a faulty c', such that $\mathbf{z}' = \mathbf{y} + c' \cdot \mathbf{s_1}$. By combining it with the correct signature $\mathbf{z} = \mathbf{y} + c \cdot \mathbf{s_1}$, they can then recover the private key $\mathbf{s_1}$ using $\mathbf{s_1} = \Delta c^{-1} \cdot \mathbf{\Delta z}$. Their work only applies to deterministic version of Dilithium signing.

Table 1 summarizes typical existing FIAs against Kyber and Dilithium, where No. faults refers to the number of fault injections required per execution, and No. executions indicates the number of algorithm executions needed for a successful attack.

Table 1: Typical existing FIAs and our attack

Attack	Type	No. faults	No. executions	Kyber			Dilithium		
				keygen	encap	decap	keygen	sign	verify
[EFGT17]	1	1	1	✗	✗	✗	✗	✓	✗
[UMB+23]	1	1	1-2k	✗	✗	✗	✗	✓	✗
[RRB+19]	1	4-11	1	✓	✓	✗	✓	✓	✗
[BBK16]	2	1	1k-2k	✗	✗	✗	✗	✓	✗
[XIU+21]	2	1	1k-3k or 1	✗	✗	✓	✗	✗	✓
[PP21]	3	1	6.5k-13k	✗	✗	✓	✗	✗	✗
[HPP21]	3	1	5k-7k	✗	✗	✓	✗	✗	✗
[DDP21]	3	1	10k-100k	✗	✗	✓	✗	✗	✗
[IMS+22]	3	1	1k-2k	✗	✗	✗	✗	✓	✗
[KPLG24]	3	1	0.5k-1k	✗	✗	✗	✗	✓	✗
[BP18]	4	1	1	✗	✗	✗	✗	✓	✗
[RYB+23]	-	1	1	✓	✓	✗	✓	✓	✓
This paper	-	1	1	✓	✓	✓	✓	✓	✓

The existing attacks have some common limitations. First, most of them are only applicable to a particular algorithm and its procedure, limiting the generality of the attacks. Type 1 does not apply to signature verification and decapsulation procedures that do not involve LWE; Types 2, 3, and 4 only apply to one or two procedures. Only Ravi et al. [RYB+23] made a commendable attempt to target the generic component NTT and were able to attack five procedures. Second, they often just support a single fault injection point, limiting the feasibility of the attacks. Attackers only have one chance to inject the target fault within a single execution, resulting in strict requirements for inducing faults in practice. Third, many works do not describe how to reliably induce the assumed faults, limiting the practicality of the attacks. There is a lack of theoretical guidance and engineering experience to mount the attacks on specific platforms quickly and effectively.

1.1 Motivation

To address the above limitations, an ideal fault attack on Kyber and Dilithium should have the following three features:

- **Feature 1:** It targets generic cryptographic primitives, applicable to all procedures of Kyber and Dilithium (In this paper, **all procedures** mean key generation, encapsulation, and decapsulation for Kyber and key generation, signing, and verification for Dilithium).
- **Feature 2:** It has multiple optional fault injection points.
- **Feature 3:** It provides a guide to induce the target faults reliably and repeatably.

For **Feature 1**, we select Keccak [BDPVA13] as the target cryptographic primitive. Keccak acts as the hash algorithm and eXtendable-Output Function (XOF) specified in the NIST standard drafts of Kyber and Dilithium [NIS23c, NIS23b]. Moreover, it is one of the most essential components and is widely employed in each procedure.

More importantly, the importance and sensitivity of the hash-like components are much higher in Kyber and Dilithium than in traditional public-key cryptography. In Kyber and Dilithium, the Keccak output is highly correlated with sensitive information, making it a valuable target for FIA. While in RSA and ECDSA, hash-like components are only used to process public information, such as the message [NIS23a]. This is the main reason why we focus on Keccek. This new exploitable target has not been explored in past research. Only Bruinderink and Pessl [BP18] presented a novel DFA by injecting faults in Keccak. However, their focus is on DFA rather than Keccak. As a result, their work is only practical on the signing procedure of the deterministic variant of Dilithium. After that, no further work has been done to thoroughly study Keccak regarding the physical security of Kyber and Dilithium.

For **Feature 2**, we use the loop-abort model to destroy the Keccak instances. Keccak contains numerous loops, providing multiple potential fault injection points to skip them, thereby disrupting the operation flow and controlling the Keccak output. Specifically, injecting faults into the iterative assignments or operations can lead to incomplete updating of the internal state, making the output set to a known value.

Furthermore, Keccak is extensively used to expand the initial random seed for the polynomial vectors in Kyber and Dilithium, such as the secret $\mathbf{s}$ and the error $\mathbf{e}$. Therefore, we can exploit the known faulty Keccak outputs to deduce or solve for secret information, thereby realizing key recovery, signature forgery, and verification bypass.

For **Feature 3**, we aim to propose a heuristic method to stably induce loop-abort faults on ARM Cortex-M devices. ARM Cortex-M embedded devices are extensively used in the real world, on which Kyber and Dilithium will definitely be implemented in the future. In

this paper, we conduct fault characterization on several Cortex-M devices and depict the relationship between fault rate and the spatial location, time offset, and pulse intensity of the Electromagnetic Fault Injection (EMFI). It further makes our proposed attacks on Kyber and Dilthium reliable and repeatable.

Note that there is a significant difference between attacking Kyber and Dilithium through exploiting Keccak (our work) and attacking Keccak itself [BGS15, LFZD16, LAFW18]. First, the targets are different. FIA on Keccak aims to recover the input, while our goal is to obtain controlled outputs and compromise Kyber and Dilithium. Second, FIA on Keccak usually requires multiple instances of Keccak with the same input, which is difficult to achieve in Kyber and Dilithium. In addition, many of the FIAs on Keccak use bitflips to incur intermediate state changes, while our work focuses on loop-abort faults. Thus, our work on Kyber and Dilithium takes an entirely different technical path from the FIAs on Keccak itself.

1.2 Contributions

This work proposes a general and practical fault attack scheme on Kyber and Dilithium. As shown in the last row of Table 1, compared to other works, our attack scheme requires only a single successful fault injection and applies to all procedures of Kyber and Dilithium. The specific contributions are summarized as follows:

- **New vulnerability:** We find that Keccak output is a critical new vulnerability in Kyber and Dilithium under FIA, since it is highly related to sensitive information, such as the secret key, the error of LWE, and the randomness of the Dilithium signature. We prove that the faulty Keccak output can lead to severe security problems. Our findings highlight the careful implementation of Keccak in Kyber and Dilithium, and guide the design of other lattice-based cryptography using hash functions like Keccak.

- **New fault attacks:** This work presents six different FIAs on Keccak under the loop-abort model, offering multiple feasible fault injection points. Any one of these attacks can set the Keccak output to a known value. We thoroughly analyze the official implementation of Kyber and Dilithium, and based on the six Keccak FIAs, we propose eight attack strategies on Kyber and seven on Dilithium, realizing key recovery, signature forgery, and verification bypass attacks. Our work is the first to apply to all the procedures of Kyber and Dilithium.

- **A guide of loop-abort fault injection:** We conduct extensive fault characterizing experiments on five commercial devices belonging to four different series (ARM Cortex-M0+, M3, M4, and M33), and provide a detailed description of the fault characteristics of these devices under different spatial positions, time offsets, and pulse intensities. Based on these characteristics, we propose a guide for stably inducing loop-abort faults using EMFI, achieving a success rate of up to 89.5% on ARM Cortex-M devices. We further design experiments to explore the interpretability of loop-abort faults and discover that instruction skipping is the cause.

- **Practical FIAs on Kyber and Dilithium:** We successfully carry out our proposed attacks on real-world devices. We first validate the Keccak attacks. All of them successfully obtain the expected faulty outputs, with success rates ranging from 12.4% to 65.1%. We further validate our complete attacks on every procedure of Kyber and Dilithium based on the official implementations. Our fault injection achieves a success rate of up to 55.1%, and upon fault occurrence, we can compromise the security of Kyber and Dilithium with a 100% probability. Our experiment

code is open-source[1].

The structure of the paper is as follows:

Section 2 introduces the attack targets Keccak, Kyber, and Dilithium. Section 3 presents our attack methods. In section 3.1, we introduce the main idea of our attack. Following this, in section 3.2, we conduct an in-depth analysis of the vulnerabilities of Keccak under loop-abort faults and present our proposed six Keccak attacks. Sections 3.3 and 3.4 present our attack methods targeting Kyber and Dilithium. Chapter 4 is dedicated to experimental validation, mainly including our fault characterizing results, fault explanation, and the practical validations of attacks for each procedure. Section 5 presents multiple countermeasures against the proposed attacks. Section 6 concludes the paper and discusses future work.

2 Background

2.1 Keccak and SHA-3

Keccak [BDPVA13] is a cryptographic hash function that has been standardized by NIST as SHA-3 [NIS15]. SHA-3 offers improved security and flexibility compared to its predecessors. Therefore, it is a preferred choice when designing cryptographic algorithms. The SHA-3 standard includes six Keccak variants, whose parameters are shown in Table 2, where d represents the output length, r represents the block size, and c represents the capacity.

Table 2: Parameters of SHA-3 variants

SHA-3 variant	d (bits)	r (bits)	c (bits)
SHA3-224	224	1152	448
SHA3-256	256	1088	512
SHA3-384	384	832	768
SHA3-512	512	576	1024
SHAKE128	any	1344	256
SHAKE256	any	1088	512

As illustrated in Figure 1, Keccak is based on sponge construction [BDPVA07]. The sponge construction has an internal state of b bits consists of two parts. The size of the first part is referred to as the *rate* (denoted r), which controls the bitrate. The size of the rest part is called the *capacity* (denoted c), which determines the maximum security level. The sponge construction mainly consists of two procedures: absorbing and squeezing. During the absorbing procedure, message blocks are bitwise XORed with the first part of the internal state. During the squeezing procedure, output blocks are read from the same subset of the state. After each of these absorbing and squeezing, the whole state is transformed using a permutation function Keccak-f. For SHA-3, Keccak-f consists of 24 rounds, operating on the internal state of 1600 bits.

There are numerous loops and iterative operations in Keccak. Firstly, the core of Keccak's diffusion and confusion, the Keccak-f permutation, is a 24-round iterative transformation. Secondly, the absorb and squeeze processes cyclically invoke Keccak-f and perform input or output operations. These operations iteratively update the internal state, protecting the input message. Additionally, the internal state of Keccak is represented by an array in implementations, and reading from or writing to this array involves loop assignments. If these loops are bypassed, Keccak cannot guarantee its security.

[1] https://anonymous.4open.science/r/Beware_of_Keccak-ADCD

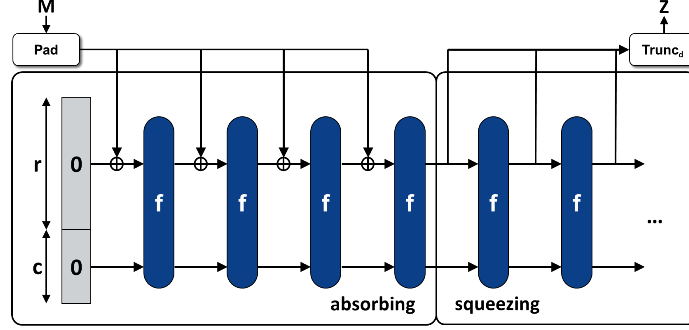


Figure 1: Sponge construction of Keccak

2.2 CRYSTALS-Kyber

Kyber [BDK+18] is a IND-CCA secure Key Encapsulation Machanism (KEM) based on the MLWE problem [LS15]. The core of Kyber KEM is the IND-CPA secure Public-Key Encryption (PKE) scheme, and the Kyber PKE is converted into the KEM using the Fujisaki-Okamoto (FO) transform [FO99]. Specifically, Kyber operates on a $k \times k$ module, donated as $R_q^{k\times k}$, where the ring $R_q = \mathbb{Z}_q[x]/(x^n + 1)$. Kyber has three variants with different security levels, namely Kyber512, Kyber768, and Kyber1024. The three variants share the same $q = 3329$ and $n = 256$, with $k = 2, 3, 4$ respectively. In this paper, polynomials, polynomial vectors, and polynomial matrices over R_q are denoted by regular font lowercase letters (e.g., a), bold lowercase letters (e.g., $\mathbf{a}$), and bold uppercase letters (e.g., $\mathbf{A}$), respectively.

Keccak plays an important role in Kyber KEM. Algorithms 1, 2, and 4 correspond to the key generation, encapsulation, and decapsulation procedures. In the key generation procedure, Keccak expands the random seed and generates secret information $\mathbf{s}$ and $\mathbf{e}$. During the encapsulation procedure, Keccak is used to hash message M, generate the random seed for encryption and the shared secret K. In the IND-CCA encryption process, as shown in Algorithm 3, Keccak is used to derive $\mathbf{r}$, $\mathbf{e_1}$, and e_2 that protect message (plaintext). During the decapsulation procedure, Keccak is mainly used to generate the shared secret. Keccak is widely applied in the generation and computation of sensitive information. Therefore, any vulnerability in it would severely compromise the security of Kyber. The NIST draft standard ML-KEM is based on Kyber [NIS23c], with some differences discussed specifically in section 3.3.4.

Algorithm 1 Kyber Key Generation $Kyber.KeyGen()$

1: $d \leftarrow \mathbb{B}^{32}$ ▷ d is 32-byte random seed
2: $(publicseed, noiseseed) \leftarrow$SHA3-512$(d)$ ▷ Expand d using SHA3-512
3: $\hat{\mathbf{A}} \leftarrow$NTT(GenerateA$(publicseed)$) ▷ Expand publicseed to $\hat{\mathbf{A}}$
4: **for** $i = 0$ to $k - 1$ **do**
5: $\quad \mathbf{s}[i] \leftarrow$Sample(SHAKE256$(noiseseed, nonce)$) ▷ Sample $\mathbf{s}$ using SHAKE256
6: **end for**
7: **for** $i = 0$ to $k - 1$ **do**
8: $\quad \mathbf{e}[i] \leftarrow$Sample(SHAKE256$(noiseseed, nonce)$) ▷ Sample $\mathbf{e}$ using SHAKE256
9: **end for**
10: $(\hat{\mathbf{s}}, \hat{\mathbf{e}}) \leftarrow$ (NTT($\mathbf{s}$),NTT($\mathbf{e}$)) ▷ Transform to NTT domain
11: $\hat{\mathbf{t}} \leftarrow \hat{\mathbf{A}} \circ \hat{\mathbf{s}} + \hat{\mathbf{e}}$ ▷ LWE instance
12: **return** $(pk = (publicseed, \hat{\mathbf{t}}), sk = (\hat{\mathbf{s}}))$

Algorithm 2 Kyber Key encapsulation $Kyber.Encap(pk)$

1: $d \leftarrow \mathbb{B}^{32}$ ▷ d is 32-byte random seed
2: $m \leftarrow$SHA3-256(d) ▷ Generate random m using SHA3-256
3: $kr = (prekey, coins) \leftarrow$SHA3-512($m$,H($pk$)) ▷ SHA3-512 generates $coins$
4: $ct \leftarrow$Kyber.Ecrypt($pk, m, coins$) ▷ Encrypt m using randomness $coins$
5: $kr \leftarrow$ ($prekey$,SHA3-256(ct)) ▷ Replace the lase 32 bits of kr
6: $K \leftarrow$SHAKE256(kr) ▷Generate shared secret K using SHAKE256
7: **return** (K, ct)

Algorithm 3 Kyber IND-CPA encryption $Kyber.Encryp(pk, m, coins)$

1: $\hat{\mathbf{A}} \leftarrow$NTT(GenerateA($publicseed$)) ▷ Recover $\hat{\mathbf{A}}$ from pk
2: **for** $i = 0$ to $k-1$ **do**
3: $\mathbf{r}[i] \leftarrow$Sample(SHAKE256($coins, nonce$)) ▷ Expand coins to $\mathbf{r}$ using SHAKE256
4: **end for**
5: **for** $i = 0$ to $k-1$ **do**
6: $\mathbf{e_1}[i] \leftarrow$Sample(SHAKE256($coins, nonce$)) ▷ Generate $\mathbf{e_1}$ using SHAKE256
7: **end for**
8: $e_2 \leftarrow$Sample(SHAKE256($coins, N$)) ▷ Generate e_2 using SHAKE256
9: $\hat{\mathbf{r}} \leftarrow$NTT($\mathbf{r}$)
10: $\mathbf{u} \leftarrow \text{NTT}^{-1}(\hat{\mathbf{A}}^T \circ \hat{\mathbf{r}}) + \mathbf{e_1}$
11: $v \leftarrow \text{NTT}^{-1}(\hat{\mathbf{t}}^T \circ \hat{\mathbf{r}}) + e_2$+Encode($m$) ▷ Calculate ciphertext $\mathbf{u}$ and v
12: **return** ct = (Compress($\mathbf{u}$),Compress(v))

Algorithm 4 Kyber Key decapsulation $Kyber.Decap(ct, sk)$

1: $m' \leftarrow$Kyber.Decrypt(ct, sk) ▷ Decrypt ct using sk
2: $kr' = (prekey', coins') \leftarrow$SHA3-512($m'$,H($pk$))
3: $ct' \leftarrow$Kyber.Ecrypt($pk, m', coins'$) ▷ Re-encrypt to get ct'
4: **if** $ct \neq ct'$ **then**
5: $K' \leftarrow$Random() ▷ K' is random value if not equals
6: **else**
7: $kr' \leftarrow$ ($prekey'$,SHA3-256(ct'))
8: $K' \leftarrow$SHAKE256(kr) ▷ Generate shared secret K' using SHAKE256
9: **end if**
10: **return** K'

2.3 CRYSTALS-Dilithium

Dilithium [DKL+18] is a Schnorr-like signature scheme based on the MLWE and the Module Short Integer Solution (MSIS) problems [KLS18]. Dilithium is standardized in the NIST draft standard for module-lattice-based digital signatures [NIS23b]. Dilithium operates on a $k \times l$ module, donated as $R_q^{k \times l}$, where the ring $R_q = \mathbb{Z}_q[x]/(x^n + 1)$. Dilithium has three variants with different security levels, namely Dilithium2, Dilithium3, and Dilithium5. The three variants share the same $q = 8380417$ and $n = 256$, with $(k, l) = (4, 4), (6, 5), (8, 7)$ respectively. Algorithms 5, 6, and 7 correspond to the key generation, signing, and verification procedures of Dilithium.

Dilithium supports both deterministic and randomized versions, with the difference in the generation of ρ' during the signing (line 2 of Algorithm 6). For the randomized version,

Algorithm 5 Dilithium Key Generation $Dilithium.KeyGen()$

1: $d \leftarrow \mathbb{B}^{32}$ ▷ d is 32-byte random seed
2: $(\rho, \rho', K) \leftarrow$SHAKE256$(d)$ ▷ Expand d using SHA3-512
3: $\hat{\mathbf{A}} \leftarrow$NTT(GenerateA$(\rho)$) ▷ Expand ρ to $\hat{\mathbf{A}}$
4: **for** $i = 0$ to $l - 1$ **do**
5: $\mathbf{s_1}[i] \leftarrow$Sample(SHAKE256$(\rho', N)$) ▷ Expand ρ' to $\mathbf{s_1}$ using SHAKE256
6: **end for**
7: **for** $i = 0$ to $l - 1$ **do**
8: $\mathbf{s_2}[i] \leftarrow$Sample(SHAKE256$(\rho', N)$) ▷ Expand ρ' to $\mathbf{s_2}$ using SHAKE256
9: **end for**
10: $\mathbf{t} \leftarrow \text{NTT}^{-1}(\hat{\mathbf{A}} \circ \text{NTT}(\mathbf{s_1})) + \mathbf{s_2}$ ▷ LWE instance
11: $(\mathbf{t_1}, \mathbf{t_0}) \leftarrow Power2Round(\mathbf{t})$
12: **return** $(pk = (\rho, \mathbf{t_1}), sk = (\rho, K, tr = \text{H}(\rho, \mathbf{t_1}), \mathbf{s_1}, \mathbf{s_2}, \mathbf{t_0}))$

rnd is a pseudorandom string, whereas for the deterministic version, it is a constant string. The randomized version generates a different signature for the same private key and message each time, providing better security.

Keccak plays a crucial role in Dilithium. In the key generation procedure, it expands the random seed and generates the private key. During the signing, Keccak generates ρ', c, and $\mathbf{y}$. Additionally, in the verification procedure, Keccak generates c' (line 3 of Algorithm 7), which is compared to determine whether the signature verification passes. The sensitivity of these operations means that vulnerabilities in Keccak could be severe.

Algorithm 6 Dilithium signing $Dilithium.Sign(sk, M)$

1: $\mu \leftarrow$SHAKE256(tr, M)
2: $\rho' \leftarrow$SHAKE256(K, rnd, μ) ▷ Generate ρ' using SHAKE256
3: $\kappa \leftarrow 0$; $(\mathbf{z}, \mathbf{h}) \leftarrow \perp$ ▷ The Fiat-Shamir with Abort loop below
4: **while** $(z, h) = \perp$ **do**
5: **for** $i = 0$ to $l - 1$ **do**
6: $\mathbf{y}[i] \leftarrow$Sample(SHAKE256(ρ', κ)) ▷ Expand ρ' into s_2 using SHAKE256
7: **end for**
8: $c \leftarrow$Sample(SHAKE256$(HighBits(\mathbf{A} \circ \mathbf{y}), \mu)$) ▷ Generate c using SHAKE256
9: $\mathbf{z} \leftarrow \mathbf{y} + c \cdot \mathbf{s_1}$ ▷ $\mathbf{z} = \mathbf{y} + c \cdot \mathbf{s_1}$
10: Make hint and Conditional Checks
11: **if** not satisfied **then**
12: $\kappa \leftarrow \kappa + l$; $(\mathbf{z}, \mathbf{h}) \leftarrow \perp$ ▷ Increase κ
13: **end if**
14: **end while** ▷ The Fiat-Shamir with Abort loop
15: **return** $\sigma = (\mathbf{z}, \mathbf{h}, c)$

Algorithm 7 Dilithium verification $Dilithium.Verify(pk, \sigma)$

1: $w_1' \leftarrow$RecoverW1(σ, pk) ▷ Recover w_1 from signature
2: $\mu \leftarrow$H(tr, M)
3: $c' \leftarrow$SHAKE256$(\mu, \mathbf{w_1'})$ ▷ Calculate c' using SHAKE256
4: **return** $(c' == c)$ and ($\mathbf{z}$ and $\mathbf{h}$ are valid) ▷ Fail if c' is wrong

3 FIA scheme on Kyber and Dilithium

3.1 Main Idea

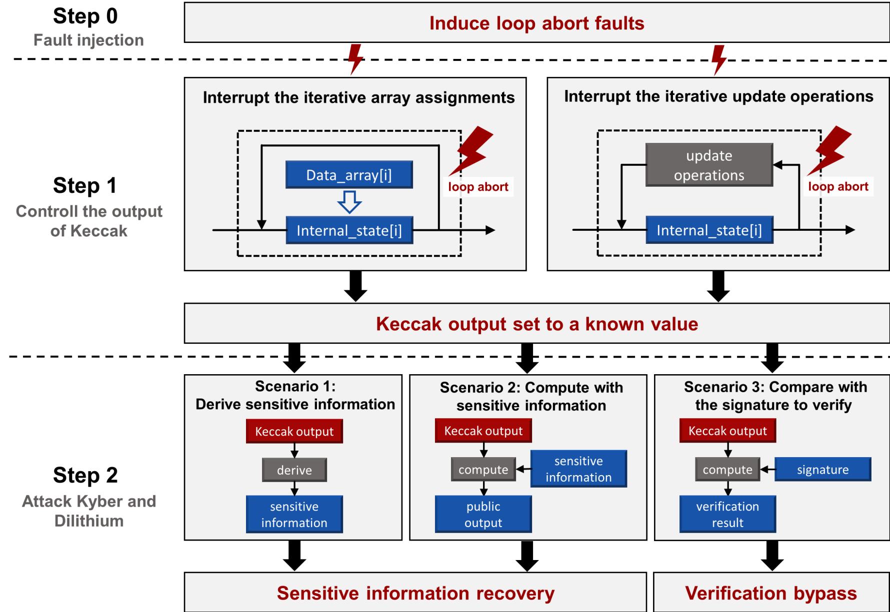


Figure 2: Main idea of our attacks on Kyber and Dilithium

As shown in Figure 2, the core idea of our work is to inject faults into the implementation of Keccak, causing its output to change to a faulty known value, and then use this ability to construct attacks that compromise the security of Kyber and Dilithium. Our attack scheme has three steps:

Step 0: Inducing loop-abort faults. Loop-abort means completely skipping or prematurely ending the loop. In Section 4, we prove that we can stably inject loop-abort faults on ARM Cortex-M devices with a high success rate. Therefore, this section (section 3) assumes that the attacker has this capability.

Step 1: For the Keccak instances, our goal is to control their output, making the output a known value. We primarily use two methods:

- **Method 1:** The internal state or the temporary values of the state are stored in the form of an array. We interrupt the iterative array assignment and leave the internal state incompletely updated. In this case, the attacker can zeroize the target array and compute the final output of Keccak.

- **Method 2:** The Keccak algorithm involves numerous iterative update operations, such as the absorb and squeeze operations and the Keccak-f permutation. By interrupting the loops of these operations, we can reduce the complexity of the Keccak transformations and thereby undermine its one-way property. In this case, the attacker can recover the input of Keccak using part of its output and finally obtain its total output, which contains sensitive information.

Step 2: Using the controlled outputs of Keccak, we can recover or change the value of the private key and other sensitive information of Kyber and Dilithium. This further enables key recovery attacks, signature forgery attacks, and verification bypass attacks. In

Kyber and Dilithium, the output of Keccak is primarily used in three different scenarios, corresponding to three attack methods.

- **Method 1:** Sensitive information, such as the expanded random seed, the secret key, and the shared secret, is directly derived from the Keccak output. This means that as long as the Keccak output is known, the corresponding sensitive information can be easily recovered.
- **Method 2:** The Keccak output is computed with sensitive information to generate a public output, such as the signature of Dilithium. Therefore, the attacker can solve for the sensitive information and recover other related secrets.
- **Method 3:** The output of Keccak is used to compare with the signature and verify the validity of the user's signature. Therefore, we can make malicious signatures pass the verification by faulting them to known values.

For **Step 1**, we propose six attacks on Keccak, offering multiple optional fault injection points (section 3.2). For **Step 2**, we propose eight attack strategies on Kyber (section 3.3) and seven on Dilithium (section 3.4), covering **all procedures** of them.

3.2 Fault Vulnerabilities of Keccak

In the official implementation of Kyber and Dilithium [pcb, pca], as well as well-known PQC algorithm libraries such as PQClean [PQC] and pqm4 [mup], Kyber and Dilithium share the same implementation of Keccak. This implementation has high representativeness and is worth analyzing.

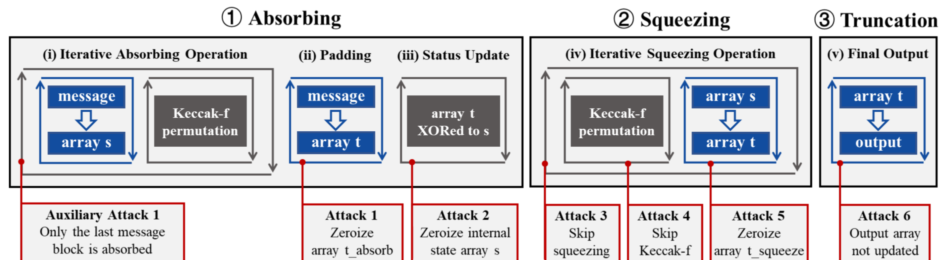


Figure 3: Main idea of our attacks on Kyber and Dilithium

Under the loop-abort model, we analyze the impact on the final output when a fault occurs in each loop of the Keccak implementation. Taking the implementation of SHA3-512 as an example, it mainly consists of three steps: absorbing, squeezing, and truncation. As shown in Figure 3, we propose six attacks and one auxiliary attack in these three steps.

3.2.1 Fault Vulnerabilities of Keccak Absorbing

Algorithm 8 is the Keccak absorbing function. At the beginning of the function, helper array t is defined to store the last padded message block. In Loop 1, the internal state array s is initialized to zero. Then, in Loop 2, the first $n-1$ message blocks are absorbed into the internal state. Each round of the loop consists of one bitwise XOR operation and one Keccak-f permutation. After that, the function handles the last message block under the Keccak padding rule. In Loop 3, array t is initialized to all zero. In loop 4, the rest of the message is copied from array m to array t. Finally, array t is XORed to the internal state array s in Loop 5.

Algorithm 8 Keccak absorbing $Keccak.absorb(m, r)$

```
1:  uint8_t t[200]                                  ▷ define helper array t
2:  for i = 0 to 25 do                              ▷ Loop 1: initialize internal state s
3:      s[i] ← 0
4:  end for
5:  for the complete message blocks do             ▷ Loop 2: absorbing, Auxiliary Attack
6:      s ←BitwiseXOR(s, messageblock)              ▷ absorb complete mesage blocks
7:      s ←Keccakf1600(s)                           ▷ Keccak-f permutation
8:  end for
9:  for i = 0 to r do                               ▷ Loop 3: initialize t
10:     t[i] ← 0
11: end for
12: for i = 0 to mlen − 1 do                        ▷ Loop 4: handle the last block, Attack 1
13:     t[i] ← m[i]                                 ▷ copy m to t
14: end for
15: t ←pad(t)                                       ▷ padding
16: for i = 0 to r/8 do                             ▷ Loop 5: update internal state s, Attack 2
17:     s[i] ← s[i] ⊕ load64(t + 8 ∗ i)             ▷ padded message XORed to s
18: end for
19: return s                                        ▷return the internal state
```

We define a condition frequently occurring in Kyber and Dilithium as the **Short Input Keccak** condition. Keccak instances satisfying this condition is more vulnerable to loop-abort faults.

Short Input Keccak: The length of the input information is smaller than the block length. This means there is only one block, and Loop 2 is not executed.

For example, in Kyber and Dilithium, Keccak instances are used to expand the random seed. The seed is typically only 32 bytes, which is smaller than the block size of all SHA-3 members and satisfies the **Short Input Keccak** condition.

We propose two attacks and one auxiliary attack on Keccak absorbing.

- **Attack 1 (zeroize-t-absorb):** We induce a loop-abort fault in Loop 4, causing the message array m to fail to be assigned to array t. The effect is equivalent to zeroizing t. As previously analyzed, if it satisfies the **Short Input Keccak** condition, after executing the absorb function, the internal state s equals the known t. We can further calculate the Keccak output using the known internal state.

- **Attack 2 (zeroize-s):** We induce a loop-abort fault in Loop 5, causing t to fail to be XORed to the internal state s, leaving s un-updated. Like Attack 1, if satisfying the **Short Input Keccak** condition, we can zeroize s and control the Keccak output.

- **Auxiliary Attack (skip-absorbing):** We induce a loop-abort fault in Loop 2, skipping the processing of the first $n-1$ message blocks. This transforms Keccak instances with any input length into the **Short Input Keccak** case. With the Auxiliary Attack, Attack 1 and Attack 2 can be carried out at all input lengths.

3.2.2 Fault Vulnerabilities of Keccak Squeezing

Algorithm 9 is the Keccak squeezing function. In each round, the Keccak-f permutation is first executed and an output of r bits is generated. The Keccak-f permutation (Loop 7) consists of 24 rounds. In the Keccak implementation in Kyber and Dilithium, two rounds are performed each time, repeating 12 times. In the following Loop 8, the first r bits of the internal state s are assigned to the buffer array t as the output of the Keccak squeezing.

Algorithm 9 Keccak squeezing $Keccak.squeeze(s, r)$

```
1:  for output blocks do                      ▷ Loop 6: squeezing loop, Attack 3
2:    for i = 0 to 11 do                      ▷ Loop 7: Keccak-f, Attack 4
3:      s ←permutation-2round(s)              ▷ 2 round × 12 times
4:    end for
5:    for i = 0 to r/8 do                     ▷ Loop 8: copy s to t, Attack 5
6:      store64(t + 8 ∗ i, s[i])              ▷ output s[i] to t 64 bits a time
7:    end for
8:    t ← t + r
9:  end for
10: return t
```

We also define a frequently occurring condition that makes attacks easier.

Short Output Keccak: The total output length is less than r bits, causing Loop 6 to execute only once, which means the Keccak squeeze function will only involve one Keccak-f permutation and one output assignment.

The fixed-length output members SHA3-224, SHA3-256, SHA3-384, and SHA3-512 have output lengths that are smaller than their corresponding r. The SHAKE128 and SHAKE256 have respective r values of 1344 and 1088, which are pretty long, so the condition is always satisfied.

We propose three attacks on the implementation of the Keccak squeezing.

- **Attack 3 (skip-squeezing):** We abort the Loop 6, causing the squeeze operations to be skipped entirely and the array t not to be updated. Though the array t is not initialized, we observe that t is zeroized in most cases. We achieve the effect of zeroizing Keccak output.
- **Attack 4 (skip-Keccak-f):** We abort Loop 7, the 24-round Keccak-f permutation. We observe two cases: completely skipping the loop and executing only once.

  In the case of a complete skip, equivalent to the Keccak-f permutation not being executed, the internal state s is not updated. Consequently, if satisfies the **Short Output Keccak** condition, the output of Keccak squeezing is equal to its input. In addition, if the input message m satisfies the **Short Input Keccak** condition, no Keccak-f permutation will be executed within the Keccak absorbing. Therefore, the final output of Keccak is the padded message, which allows us to obtain the input through the output of Keccak, thus compromising the one-way property.

  If loop executed only once, the output of Keccak will expose the input since only two rounds of permutation are performed. We can reverse the 2-round Keccak-f and recover the Keccak input from the output (detailed in Section 3.2.4).
- **Attack 5 (zeroize-t-squeeze):** We induce a loop-abort fault in Loop 8, causing the internal state array s to fail to be assigned to the output buffer array t. The array t remains un-updated, which results in the zeroization of the Keccak output.

3.2.3 Fault Vulnerabilities of the Keccak Output Truncation process

As shown in Algorithm 10, the implementation of the Keccak output truncation process is straightforward. In Loop 9, in the case of SHA3-512, the first 64 elements of array t are assigned to the output array, which is the final output of Keccak.

- **Attack 6 (skip-output):** We induce a loop-abort fault in Loop 9, causing array t to fail to be assigned to the final output array. Consequently, in some cases, the

final output remains all zero. In other cases in Kyber and Dilithium, the output array exposes random seeds previously stored in the same address.

Algorithm 10 Keccak truncation $Keccak.trunc(t, d)$

```
1: for i = 0 to d do                    ▷ Loop 9: generate output of length d, Attack 9
2:     output[i] ← t[i]                 ▷ copy squeezing output t to the final output
3: end for
4: return output
```

3.2.4 Solving the Input of 2-round Keccak-f

In this subsection, we describe the method for reversing the 2-round Keccak-f in Attack 4. This part is not the focus of our work, and we just provide a feasible solution.

Our assumptions and goal are as follows: First, assume that we have part of the output of Keccak. For example, in the key generation procedure of Kyber and Dilithium, the first 256 bits of the output of a SHA3-512 instance constitute the public key, which an attacker can easily obtain. Second, assume that we have successfully injected the fault described in the second case of Attack 4. In addition, assume that we also know the length of the input (which is less than r bits). We aim to recover the complete input and output that may contain sensitive data.

Our approach has three steps. First, based on the Keccak-f permutation operations, we define new intermediate variables and construct the boolean equations for the relationship between each step's input and output. Second, we set up the equation system for each step and simplify it into conjunctive normal form (CNF). Third, we use a SAT solver to solve the equation system and obtain the final result.

For the first two steps, we provide a tool to extract the CNF for Keccak, which is included in our open-source codebase. For the third step, we use miniSAT, a minimalistic, open-source SAT solver [ES03]. We can recover up to 64 bits of the input from a 128-bit faulty Keccak output in one minute on a PC.

Table 3: Conclusion of our attacks on Keccak

Attack	Description	Effect
Auxiliary Attack	Abort the loop that processes message blocks during absorbing	Only the last block is absorbed; Satisfy the Short Input Keccak case;
Attack 1	Abort the loop that assigns the last message block to array t	Zeroize helper array t; Set the output to a fixed value;
Attack 2	Abort the loop that updates the internal state s during absorbing	Zeroize internal state array s; Set the output to a fixed value;
Attack 3	Abort the outer loop of squeezing operations	Skip the whole squeeze function; Zeroize the Keccak output;
Attack 4	Abort the 24-round Keccak-f permutation loop	Solve and recover input from output; Compromise the one-way property;
Attack 5	Abort the loop that assigns the output block to buffer array t	Zeroize buffer array t; Zeroize the Keccak output;
Attack 6	Abort the loop that assigns buffer array t to the final output array	Keccak output not updated; Output array retains previous value;

3.2.5 Conclusion of Our Attacks on Keccak

We propose six fault attacks on Keccak under the loop-abort model. The attacker only needs to implement one of the attacks, injecting a fault at one of the six available fault injection points, to set the Keccak output to a known value. The attacker can also inject faults at multiple points in a single execution to maximize the attack success rate. Specifically, our attack description and effects are as shown in Table 3.

3.2.6 Fault Vulnerabilities of Keccak in Other Cryptographic Libraries

We analyzed the Keccak implementations in multiple widely used open-source cryptographic libraries. We selected mainstream post-quantum cryptography algorithm libraries like PQClean [PQC] and pqm4 [mup] and other general-purpose cryptographic libraries such as OpenSSL [ope], Mbed TLS [mbe], and wolfSSL [wI]. The results, detailed in Table 4, indicate that most of the proposed attacks still apply to other Keccak implementations, demonstrating that this is a generic vulnerability in Keccak.

Table 4: Scalability of our attacks on other Keccak implementations

Implementation	Auxiliary Attack	Attack 1	Attack 2	Attack 3	Attack 4	Attack 5	Attack 6
Official implementation of Kyber and Dilithium	✓	✓	✓	✓	✓	✓	✓
PQClean and pqm4	✓	✓	✓	✓	✓	✓	✓
OpenSSL 3.3.0	✓	✗	✓	✓	✓	✓	✓
Mbed TLS 3.6.0 LTS	✓	✗	✓	✓	✓	✓	✓
wolfSSL 5.7.0	✓	✗	✓	✓	✓	✗	✗

Some implementation details in these cryptographic algorithm libraries pose challenges for our attacks. The first is using memory copy instead of loop-based array assignments, such as the XMEMCOPY used in wolfSSL, which affects the implementation of our attacks 1, 5, and 6. The second is the reduction of data assignment between intermediate state arrays. For example, in OpenSSL and Mbed TLS, the squeeze output is directly assigned to the target array instead of first being stored in an internal state, which causes our attacks 5 and 6 to be combined into a single attack in such cases (the blue checkmarks).

We also found that in Mbed TLS and OpenSSL, the Keccak-f implementation executes only one round of permutation each time. This means that the second case of Attack 4 only needs to reverse one round of Keccak-f instead of two.

3.3 Kyber Attacking Strategies

In this section, we analyze the vulnerabilities of Kyber under our Keccak attacks. We analyze the official implementation, which has also been adopted in well-known post-quantum cryptographic algorithm libraries PQClean and pqm4. We propose eight attack strategies on the implementation of the Kyber key generation, encapsulation, and decapsulation procedures. Each attack can lead to key recovery or shared secret leakage. For the first attack, we describe in detail the method of using the six Keccak attacks we proposed to undermine the security of Kyber. For the other attacks, we adopt a more straightforward expression for brevity. We abstract the attacker's capability as being able to set the output of Keccak to a faulty known value through fault injection.

Table 5 shows the attack strategies we proposed against Kyber. It provides the targeted procedure, the required number of faults and executions (on Kyber768), and a brief description of each strategy.

Table 5: Map of our attack strategies on Kyber

Attack	Procedure	Description	Faults	Executions	Section
Kyber Attack 1	keygen	Attack random seed expansion Private key recovery	1	1	3.3.1
Kyber Attack 2	keygen	Attack sampling and control **s** Private key recovery	3	1	3.3.1
Kyber Attack 3	keygen	Attack sampling and control **e** Private key recovery	3	1	3.3.1
Kyber Attack 4	encap	Control randomness m Shared key recovery	1	1	3.3.2
Kyber Attack 5	encap	Directly control shared key Shared key recovery	1	1	3.3.2
Kyber Attack 6	encap	Control *coins* of encryption Shared key recovery	1	1	3.3.2
Kyber Attack 7	encap	Attack sampling and control **r** Shared key recovery	3	1	3.3.2
Kyber Attack 8	decap	Directly control shared key Shared key recovery	1	1	3.3.3

To help understand the attacks we propose in each procedure, we describe a key exchange protocol based on the IND-CCA secure Kyber KEM. As shown in Figure 4, Alice and Bob hope to exchange a shared key K. First, Alice generates the public key pk and private key sk and sends pk to Bob. Then, Bob generates a random message m and encrypts it to obtain the ciphertext ct. After that, Bob uses pk, m, and ct to calculate the shared key K. Afterwards, Bob sends ct to Alice. If correct, Alice calculates shared key K with m, pk, and ct. In this way, Alice and Bob finally share the same K.

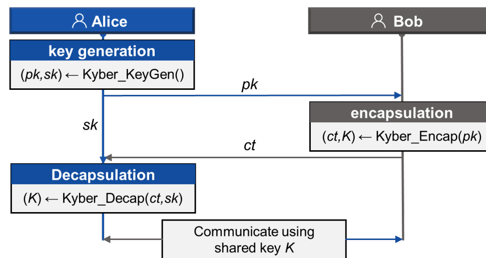


Figure 4: A Kyber-KEM key exchange protocal

3.3.1 Attacking the Key Generation Procedure of Kyber

According to Algorithm 1, the implementation of the Kyber key generation procedure can be represented by Figure 6. First, a 32-byte random string is generated by the *randombytes* function. The implementation of this function varies for different operating systems or bare metal environments, so we do not consider its security. The 32-byte random seed is stored in the first half of a 64-byte buffer. Then, the random seed is processed by SHA3-512, expanding it into a 64-byte random number stored in the same buffer. The first half of the random numbers, the *publicseed*, generates matrix **A**. The

second half, the *noiseseed*, generates the secret key $\mathbf{s}$ and error $\mathbf{e}$. This procedure can be described as follows: For each element of the polynomial vectors $\mathbf{s}$ and $\mathbf{e}$, the first step is to hash the *noiseseed* and nonce using SHAKE256 (prf function) and produce a 128-byte output. This output is used for sampling to generate random polynomial coefficients following a central binomial distribution (*cbd* function). The nonce is initialized to 0 and incremented by 1 each time a polynomial is generated.

Figure 5 illustrates our attack method during the key generation procedure. By injecting faults, the attacker can obtain the value of the private key sk either by solving for it from the public key or directly calculating its value using known Keccak output. Furthermore, with sk, one can decrypt the ciphertext ct and generate the shared key K. Finally, the attacker can use K to decrypt and eavesdrop on the communication between Alice and Bob. Based on this method, we propose three different attack strategies. Our specific attack strategies are shown in Figure 6.

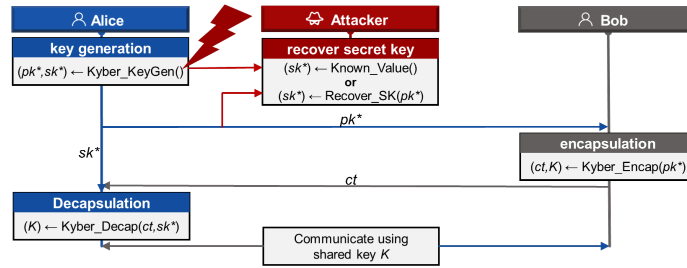


Figure 5: Attack method during the key generation procedure

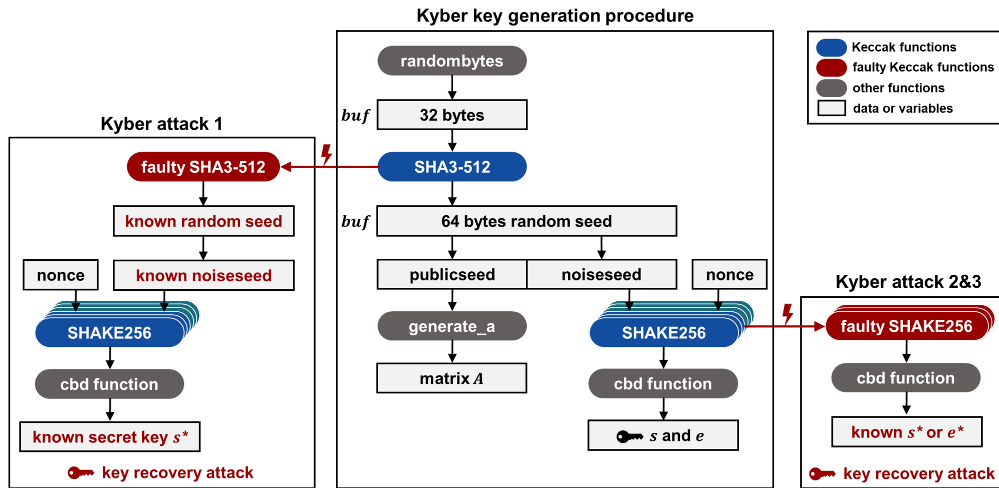


Figure 6: Kyber key generation procedure and our attacks

- **Kyber Attack 1 (control-random-seed):** We apply the six proposed Keccak attacks to the SHA3-512 instance that generates *publicseed* and *noiseseed*. These attacks set the output of the target SHA3-512 instance to a known value, exposing the expanded random seed. The attacker then can recover the secret key by sampling the known *noiseseed*. This SHA3-512 instance has a 32-byte input and a 64-byte

output, satisfying the **Short Keccak** (**Short Input Keccak** and **Short Output Keccak**) conditions, and making all of our Keccak attacks feasibile.

For Attacks 1, 2, 3, and 5 (**zeroize-t-absorb**, **zeroize-s**, **skip-squeezing**, **zeroize-t-squeeze**), the output of the SHA3-512 instance, including the *noiseseed*, is set to a fixed, known value. Consequently, we can recover the private key.

For Attack 6 (**skip-output**), the output array is not updated. According to our previous analysis, the output retains the value previously stored in its address, i.e., the initial 32-byte random seed d. This results in the last 32 bytes of the expanded random number (*noiseseed*) being all zero, leading to a known private key.

For the first case of Keccak Attack 4 (**skip-Keccak-f**, complete skip), the output equals the padded input. Since the input is a 32-byte random seed, the last 32 bytes of the output can be determined by the padding rules and is a fixed known value.

For the second case of Keccak Attack 4 (**skip-Keccak-f**, two rounds), the input is only processed through padding and a 2-round Keccak-f permutation. The first 32 bytes of the output, *publicseed*, can be obtained from the public key. We solve the 32-byte input random seed using the method we described in 3.2.4. Then we simulate the faulted SHA3-512 with 2-round Keccak-f permutation and recover the total 64-byte output, including the *noiseseed*.

- **Kyber Attack 2 (control-s):** As shown in Figure 6, we perform fault attacks on multiple SHAKE256 instances (3 instances for Kyber768) used to generate the polynomial vector of secret key $\mathbf{s}$. This allows us to set the prf function (SHAKE256 instances) outputs to known values and determine the secret key, achieving key recovery attacks. These SHAKE256 instances have a 32-byte input and a 128-byte output, satisfying the **Short Keccak** conditions.

- **Kyber Attack 3 (control-e):** Like Kyber Attack 2, we perform fault attacks on the SHAKE256 instances (3 instances for Kyber768) used to generate $\mathbf{e}$. We get a weak LWE instance $\mathbf{t}^* = \mathbf{A} \cdot \mathbf{s} + \mathbf{e}^*$ where $\mathbf{e}^*$ is a faulty known polynomial vector. Because both $\mathbf{t}^*$ and $\mathbf{A}$ (the public key) are known, we can solve for the secret key and achieve key recovery attacks. These SHAKE256 instances satisfy the **Short Keccak** conditions.

3.3.2 Attacking the Encapsulation Procedure of Kyber

Algorithm 2 shows the implementation of the IND-CCA secure Kyber KEM based on the FO transformation. Figure 7 shows the Kyber KEM encapsulation procedure. First, a 32-byte random number is generated using the *randombytes* function. It is then hashed by a SHA3-256 to generate the message m for the IND-CPA encryption. Together with the hash of the public key, it is hashed using SHA3-512 to generate a 64-byte kr. The last 32 bytes of kr are the random seed *coins* for the IND-CPA secure encryption. The output of the encryption, the ciphertext *ct*, is hashed using SHA3-256 to replace the last 32 bytes of kr. The new kr is then hashed by SHAKE256 to derive the shared key K.

The IND-CPA secure Kyber encryption is shown in Algorithm 3 and is illustrated in Figure 8. The encryption starts by expanding matrix $\mathbf{A}$ and extracting $\mathbf{t}$ from the publicseed. After that, $\mathbf{r}$, $\mathbf{e_1}$, and e_2 are sampled from seed *coins*. The sampling process is the same as the sampling in the key generation, which includes the prf function using SHAKE256 and the *cbd* function that generates central binomial distributed polynomial coefficients. Afterward, $\mathbf{u}$ and v are calculated using $\mathbf{u} = \mathbf{A}^T \cdot \mathbf{r} + \mathbf{e_1}$ and $v = \mathbf{t}^T \cdot \mathbf{r} + encode(m)$, and compressed $(\mathbf{u}, v)$ is the final ciphertext *ct*.

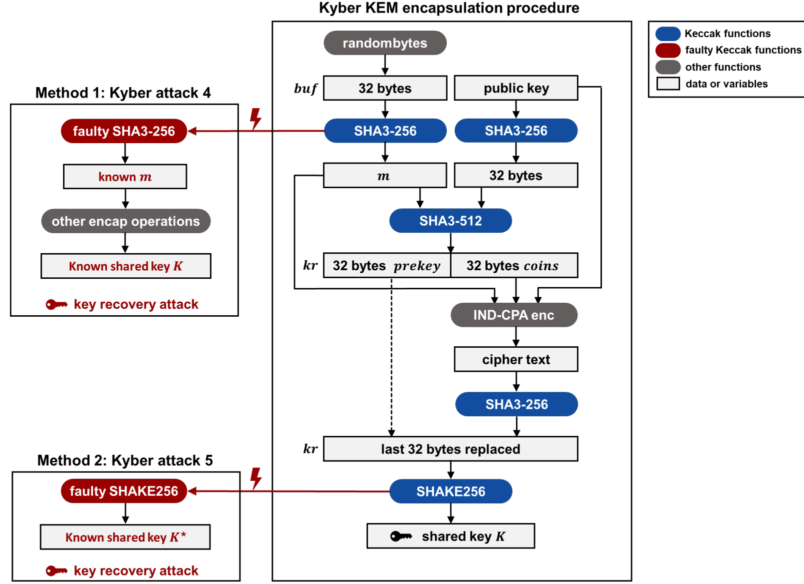


Figure 7: Kyber encapsulation procedure and our attacks

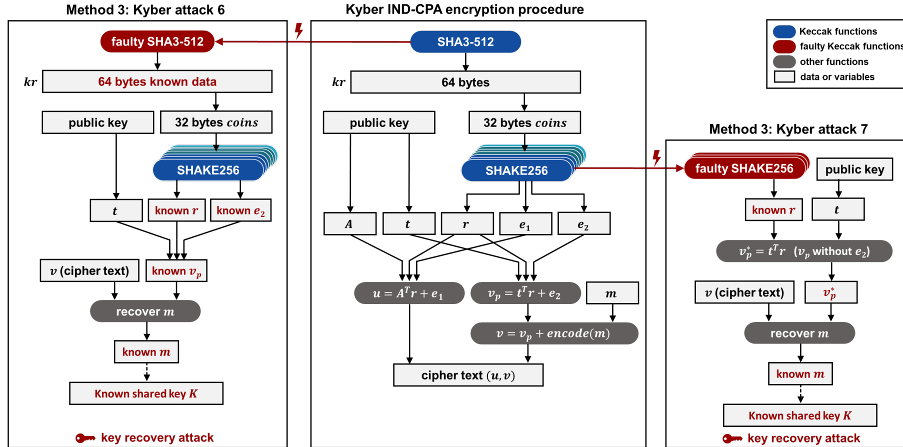
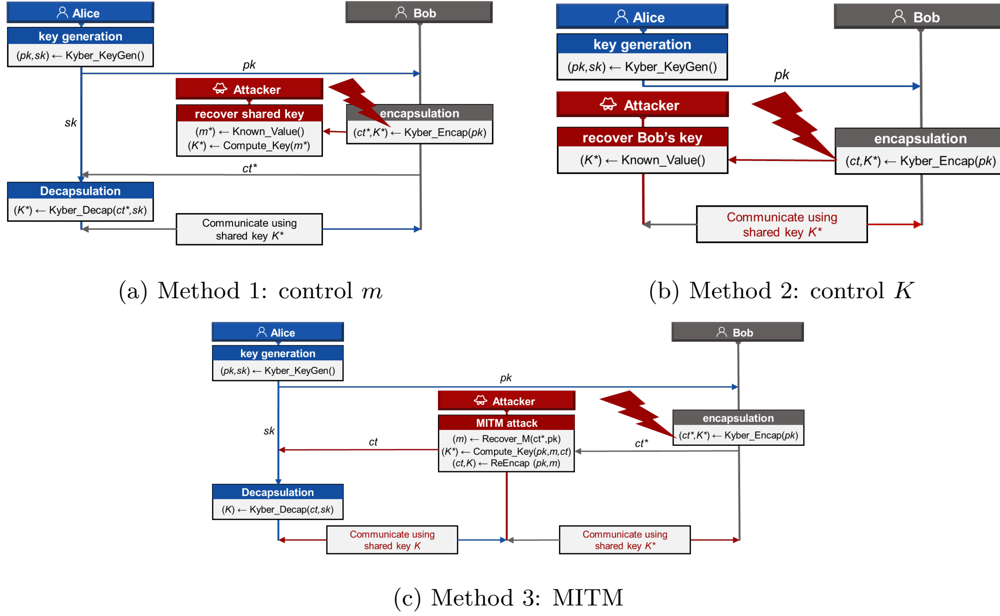


Figure 8: Kyber IND-CPA secure encryption procedure and our attacks

Figure 9 shows the three methods for attacks during the encapsulation procedure.

- **Method 1:** Control the value of the message and calculate the corresponding ciphertext ct and shared key K, achieving key recovery attacks.
- **Method 2:** Set the value of the shared key to a faulty known K^*. Then the attacker can impersonate Alice and communicate with Bob using the faulty K^*.
- **Method 3:** Inject faults during the generation of ct, allowing the attacker to solve m from the faulted ct, thereby recovering the shared key. To pass the re-encryption, we adopt the fault-assisted Man-In-The-Middle (MITM) attack scheme proposed by [RYB+23]. Suppose an attacker can intecept the communication between Alice and Bob. First, the attacker recovers m from the faulted ciphertext ct^*. Then, Bob's



(a) Method 1: control m

(b) Method 2: control K

(c) Method 3: MITM

Figure 9: Attack methods during the encapsulation procedure

shared key K^* can be calculated using the knowledge of ct^*, pk, and m. With m and pk, the attacker can recover the valid ct and calculate the corresponding K by executing the encapsulation procedure. The attacker then sends the valid ciphertext ct to Alice, who decapsulates it and gets the same K. Finally, the attacker can decrypt all communication between Alice and Bob using K and K^*.

We propose our fourth fault attack based on Method 1.

- **Kyber Attack 4 (control-m):** As shown in Figure 7, we perform fault attack on the SHA3-256 instance used to generate the message m and set it to a known value. The input and output are both 32 bytes, satisfying the **Short Keccak** conditions. Given m, the attacker can conduct the remaining encapsulation process and obtain the final shared key K, thereby achieving key recovery attacks.

We propose our fifth fault attack based on Method 2.

- **Kyber Attack 5 (control-K-encap):** As shown in Figure 7, we perform fault attacks on the SHAKE256 instance that generates the final shared key K. This SHAKE256 has a 64-byte input and a 32-byte output, satisfying the **Short Keccak** conditions. The attacker can directly set the shared key to a faulty known value.

We propose two attacks based on Method 3.

- **Kyber Attack 6 (control-coins):** As shown in Figure 8, we perform fault attacks on the SHA3-256 instance that generates kr. The input and output are 64 bytes, satisfying the **Short Keccak** conditions. The output kr, including the random seed $coins$, is fixed and known. Therefore, the attacker can calculate $\mathbf{r}$, $\mathbf{e_1}$, and e_2 derived from $coins$. With the known values of ct (including v), $\mathbf{r}$, $\mathbf{e_1}$, and $\mathbf{t}$ (part of the public key), the attacker can recover m through the following equation.

$$m = \text{Compress}(\text{Decompress}_q(v, d_v) - \mathbf{t}^T \cdot \mathbf{r} - e_2, 1) \tag{1}$$

Kyber encryption uses *Compress* functions to reduce the length of the ciphertext. Compression and decompression introduce a certain amount of error, but this does not affect the correctness of the decryption.

The correctness of this equation can be proven as follows. According to the correctness of Kyber KEM [BDK+18], there is:

$$\|w\|_\infty < \lceil \frac{q}{4} \rfloor \implies m = \text{Compress}(w + m \cdot \left\lceil \frac{q}{2} \right\rfloor, 1) \tag{2}$$

We also have the value of v without being compressed is: $v = \mathbf{t}^T \cdot \mathbf{r} + m \cdot \lceil \frac{q}{2} \rfloor$ Substitute it into equation (1), which gives the following equation, where c_v is the decompression error.

$$m = \text{Compress}(c_v + m \cdot \left\lceil \frac{q}{2} \right\rfloor, 1) \tag{3}$$

We compare c_v with the the error from the correct Kyber decryption process. It is obvious that:

$$\|c_v\|_\infty \leqslant \|\mathbf{e}^T \mathbf{r} + e_2 + c_v - \mathbf{s}^T \mathbf{e_1} + \mathbf{c}_t^T \mathbf{r} - \mathbf{s}^T \mathbf{c}_u\|_\infty \tag{4}$$

Where c_u, c_v, and c_t are the faults generated by compression and decompression. According to Kyber's correctness [BDK+18], there is:

$$Pr[\|\mathbf{e}^T \mathbf{r} + e_2 + c_v - \mathbf{s}^T \mathbf{e_1} + \mathbf{c}_t^T \mathbf{r} - \mathbf{s}^T \mathbf{c}_u\|_\infty \leqslant \lceil \frac{q}{4} \rfloor] \geqslant 1 - 2^{-142} \tag{5}$$

According to Equations (4) and (5), we obtain:

$$Pr[\|c_v\|_\infty \leqslant \lceil \frac{q}{4} \rfloor] \geqslant 1 - 2^{-142} \tag{6}$$

By combining Equation (6) with Equation (2) and (3), the proof is complete.

- **Kyber Attack 7 (control-r):** As shown in Figure 8, we perform our Keccak attacks on multiple SHAKE256 instances (3 instances for Kyber768) used to generate the polynomial vector $\mathbf{r}$. This allows attackers to control the output values of the prf function and determine the value of $\mathbf{r}$. With the known values of v, $\mathbf{r}$, and $\mathbf{t}$, the attacker can recover m using the following equation.

$$m = \text{Compress}(\text{Decompress}_q(v, d_v) - \mathbf{t}^T \mathbf{r}, 1) \tag{7}$$

Similar to Kyber Attack 6, the following inequality holds with very high probability, which proves the correctness of the above equation.

$$\|c_v + e_2\|_\infty \leqslant \|\mathbf{e}^T \mathbf{r} + e_2 + c_v - \mathbf{s}^T \mathbf{e_1} + \mathbf{c}_t^T \mathbf{r} - \mathbf{s}^T \mathbf{c}_u\|_\infty \leqslant \lceil \frac{q}{4} \rfloor \tag{8}$$

3.3.3 Attacking the Decapsulation Procedure of Kyber

According to Algorithm 4, the implementation of the Kyber decapsulation procedure can be represented by Figure 11. First, the algorithm uses the private key and the received ciphertext to decrypt the message, obtaining m'. Next, to ensure the IND-CCA security of the Kyber KEM, the message m' is re-encrypted to produce a ciphertext, ct'. This re-encryption process is the same as the encapsulation procedure. Finally, compare the newly recovered ciphertext ct' to the received ciphertext. If they match, replace the last 32 bytes of kr and use the SHAKE256 hash function to derive the final shared key K.

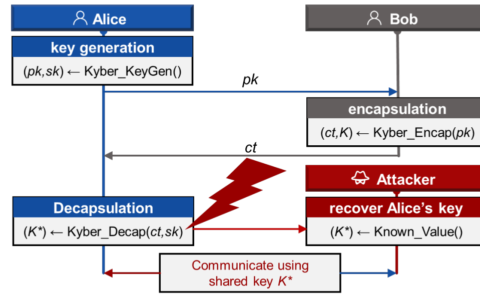


Figure 10: Attack method during the decapsulation procedure

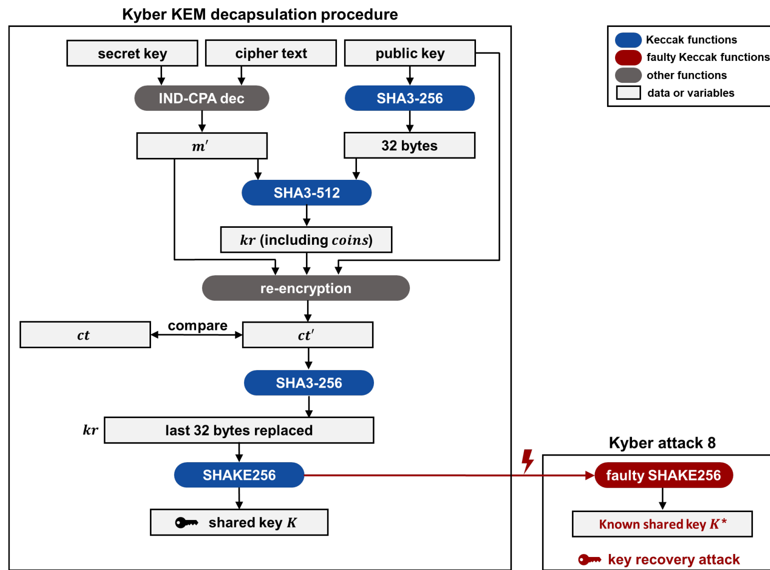


Figure 11: Kyber decapsulation procedure and our attack

Figure 10 illustrates our attack method during the decapsulation procedure. The attacker injects faults to generate a faulty known shared key K^*, enabling key recovery attacks. Subsequently, the attacker can masquerade as Bob and communicate with Alice using K^*. Our specific attack process is shown in Figure 11. Based on this method, we propose our eighth attack.

- **Kyber Attack 8 (control-K-decap):** This attack is similar to Kyber Attack 5 (**control-K-encap**). This attack targets the SHAKE256 instance that generates shared key K, This instance also satisfies the **Short Keccak** conditions. By controlling the Keccak output, the attacker can directly set the shared key to a faulty known value, enabling key recovery attacks.

3.3.4 Analyzing the NIST Draft FIPS 203 Version of Kyber

NIST standardized Kyber as ML-KEM in Draft FIPS 203 [NIS23c]. Since fault attacks depend on specific implementations, and ML-KEM has yet to be widely implemented, the above analysis is focused on CRYSTALS-Kyber. We now discuss the differences in the vulnerability of ML-KEM and Kyber under our attacks. The ML-KEM affected three of

our eight Kyber attacks, namely Kyber Attack 4, 5, and 8 (**control-m, control-K-encap, control-K-decap**). We also adapt these attacks accordingly to fit the ML-KEM.

ML-KEM made two major modifications to Kyber. First, the initial randomness m is no longer hashed but is directly generated by the random number generator. Second, ML-KEM use a different variant of the FO transform [st23a, st23b]. The encapsulation algorithm no longer includes a hash of the ciphertext when deriving the shared secret. The decapsulation algorithm has also been updated to match this change.

As shown in Figure 12, during the encapsulation procedure, our proposed Kyber Attack 4 and 5 (**control-m, control-K-encap**) are no longer applicable. However, we can migrate the target of Kyber Attack 5 to the SHA3-512 instance generating kr and merge it with Kyber Attack 6 (**control-coins**), controlling the value of the shared secret K.

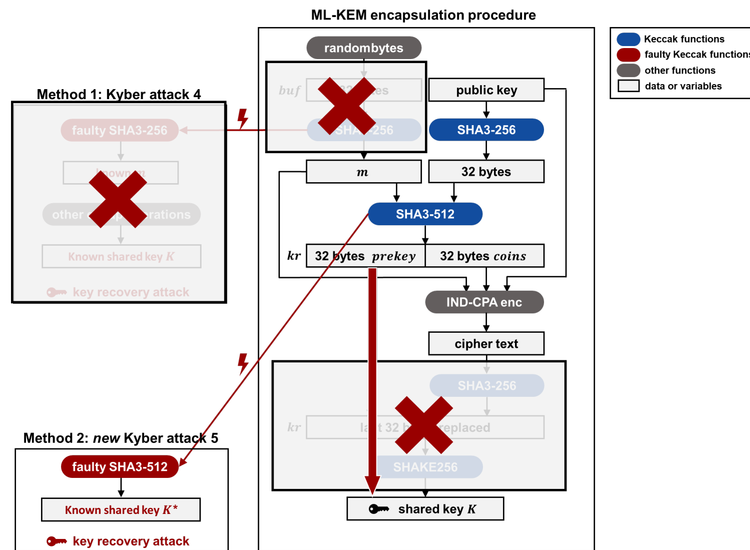


Figure 12: Impact of ML-KEM on our attacks during the encapsulation procedure

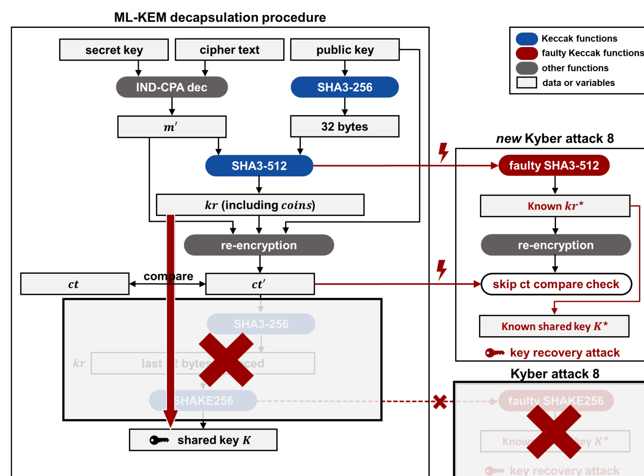


Figure 13: Impact of ML-KEM on our attacks during the decapsulation procedure

As shown in Figure 13, in the decapsulation procedure, the modification to the FO transform has rendered Kyber Attack 8 (**control-K-decap**) ineffective. Kyber Attack 8 can be migrated to the SHA3-512 instance that generates kr. However, if the value of kr changes, the generated ct' will fail to pass the comparison check. ML-KEM includes the shared secret in the check scope, making it more secure. As a countermeasure, we can utilize the attack proposed by Xagawa et al. [XIU+21] to skip the ciphertext comparison, allowing our new attack to bypass the verification while controlling the shared secret.

3.4 Dilithium Attacking Strategies

In this section, we analyze the vulnerabilities of Dilithium under our Keccak attacks based on the official implementation [pca]. Kyber and Dilithium share the same Keccak implementation, so our Keccak attacks are also applicable to the instances in Dilithium. Our Keccak attacks allow the attackers to set the output of Keccak to a faulty known value through fault injection. Based on this assumption, we conduct detailed analyses of the Dilithium key generation, the signing, and the verification procedures. We further propose seven attack strategies, including key recovery, forgery, and verification bypass attacks. Our work applies to both deterministic and randomized versions.

We summarize our Dilithium attack strategies as Table 6. It provides the targeted procedure, the required number of faults and executions (on Dilithium2), and a brief description of each strategy.

Table 6: Map of our attack strategies on Dilithium

Attack	Procedure	Description	Faults	Executions	Section
Dilithium Attack 1	keygen	Attack random seed expansion Key recovery, signature forgery	1	1	3.4.1
Dilithium Attack 2	keygen	Attack sampling and control $\mathbf{s_1}$ Key recovery, signature forgery	4	1	3.4.1
Dilithium Attack 3	keygen	Attack sampling and control $\mathbf{s_2}$ Key recovery, signature forgery	4	1	3.4.1
Dilithium Attack 4	signing	Faulty signature with known $\mathbf{y}$ Key recovery, signature forgery	1	1	3.4.2
Dilithium Attack 5	signing	Control coefficients of $\mathbf{y}$ Key recovery, signature forgery	1	$\geqslant 4$	3.4.2
Dilithium Attack 6	signing	DFA using controlled c Key recovery, signature forgery	1	1	3.4.2
Dilithium Attack 7	verification	Control c' to pass comparison Verification bypass	1	1	3.4.3

Figure 14 illustrates an interactive process based on Dilithium. First, Alice generates a public-private key pair and sends the public key to Bob. Then, Alice signs a message m and sends the message and the signature to Bob. Finally, Bob verifies the signature, and if the verification passes, it proves that the message originated from Alice.

3.4.1 Attacking the Key Generation Procedure of Dilithium

According to Algorithm 5, the implementation of the Dilithium key generation procedure can be represented by figure 16. First, the *randombytes* function generates a 32-byte initial random seed. It is then expanded by SHAKE256 into a 128-byte random string,

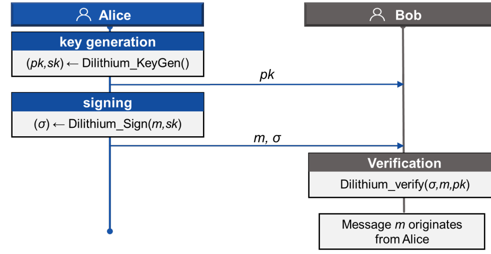


Figure 14: An interactive procedure based on Dilithium

which is stored at the same address as the seed. The first 32 bytes of the string, namely ρ, are used to generate matrix A. The following 64 bytes, namely ρ', are used to generate the secret $\mathbf{s_1}$ and error $\mathbf{s_2}$. The last 32 bytes are used as nonce.

The method of our attacks during the key generation procedure is illustrated in Figure 15. By injecting faults, the attacker can obtain the value of the private key $\mathbf{s_1}$ either by solving from the public key or directly calculating its value using known Keccak output. According to [BP18, RJH+19], knowing only $\mathbf{s_1}$ is sufficient to forge a Dilithium signature, signing any message they want. As shown in Figure 16, we propose three different attacks based on this method.

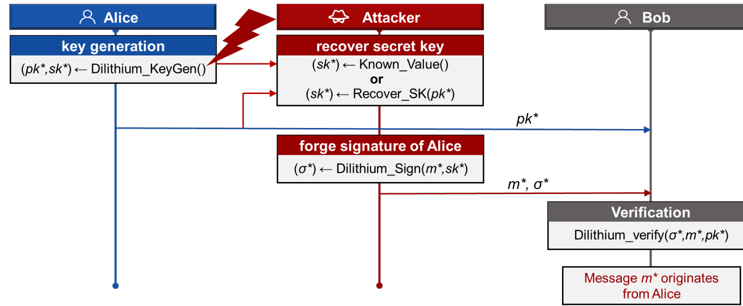


Figure 15: Attack method during the key generation procedure

- **Dilithium Attack 1 (control-random-seed):** We perform fault attacks on the SHAKE256 instance that expands the random seeds. This instance has a 32-byte input and a 128-byte output, satisfying the **Short Keccak** conditions. The attacker performs the Keccak attacks and makes the output of this instance a known value. Therefore, the attacker can get to know the value of ρ, $\mathbf{s_1}$, $\mathbf{s_2}$, and *key*. Furthermore, the attacker can compute the values of tr and t_0 and recover the entire secret key, thereby forging signatures.

- **Dilithium Attack 2 and 3 (control-s, control-e)**: We perform fault attacks on multiple SHAKE256 instances (4 instances for Dilithium2) that generate the polynomial vector of secret key $\mathbf{s_1}$ (Dilithium Attack 2) or the error $\mathbf{s_2}$ (Dilithium Attack 3). For Dilithium Attack 2, the attacker can get the secret key $\mathbf{s_1}$ directly. For Dilithium Attack 3, the attacker obtains $\mathbf{s_2}$, resulting in a weak LWE instance $\mathbf{t}^* = \mathbf{A} \cdot \mathbf{s_1} + \mathbf{s_2}^*$. This allows them to recover the private key $\mathbf{s_1}$ from the public key.

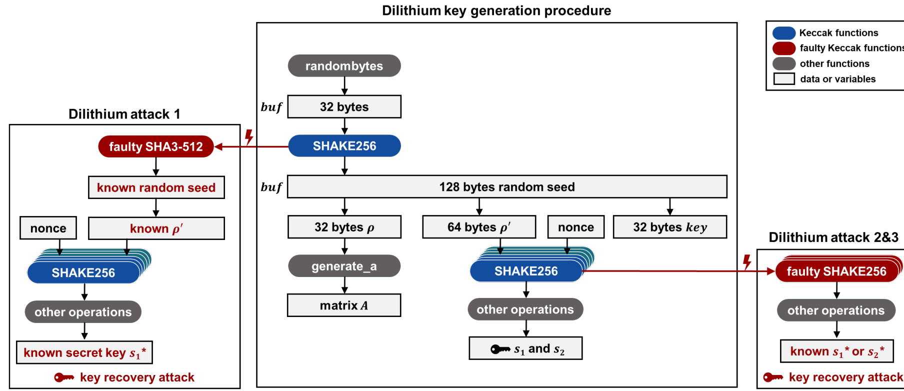
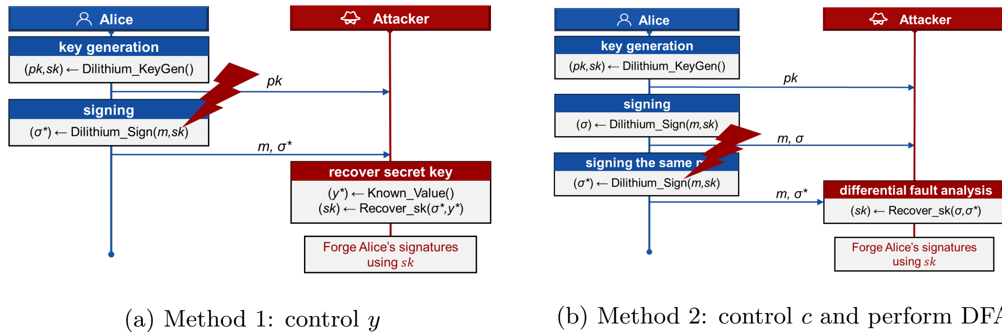


Figure 16: Dilithium key generation procedure and our attacks

3.4.2 Attacking the Signing Procedure of Dilithium

Algorithm 6 shows the implementation of the signing procedure of Dilithium, and its implementation logic is shown in Figure 18. The Dilithium signing procedure is based on the Fiat-Shamir with Aborts scheme [Lyu09, Lyu12], which continuously generates and rejects signatures until the conditions are met. First, the algorithm hashes tr and m to generate μ. For the deterministic version, ρ' is generated by hasing μ and key with SHAKE256. For the randomized version, an additional randomness rnd is involved. The polynomial vector $\mathbf{y}$ of length L (one of the parameters of Dilithium) is generated afterward. Each polynomial in the vector is generated using the SHAKE256 hash function with ρ' and a nonce κ that increments by one each time. After that, the high bits of $\mathbf{A} \cdot \mathbf{y}$ and μ are hashed using SHAKE256 and sampled to a ball, generating c. Subsequently, $\mathbf{z}$ is computed by $\mathbf{z} = \mathbf{y} + \mathbf{s_1} \cdot c$. Then, the algorithm generates a hint $\mathbf{h}$. A series of conditions check the signature $\sigma = (\mathbf{z}, \mathbf{h}, c)$. If the conditions are not met, the value of κ increases and the algorithm recalculates the signature starting from the generation of $\mathbf{y}$.

Our approach injects faults to create a faulty signature that exposes knowledge about secret key $\mathbf{s_1}$, enabling key recovery attacks and forgery attacks. We propose two methods and three attack strategies, applicable to both the **deterministic and randomized versions** of Dilithium.



(a) Method 1: control y

(b) Method 2: control c and perform DFA

Figure 17: Attack methods during the signing procedure

Figure 17 illustrates our attack methods during the signing procedure.

- **Method 1:** We generate an entirely or partially known faulty $\mathbf{y}$, donated as $\mathbf{y}^*$. For $\mathbf{z}^* = \mathbf{y}^* + c \cdot \mathbf{s_1}$, where $\mathbf{z}^*$ and c are known because they are parts of the signature.

If the $\mathbf{y}^*$ is also known, it can expose the secret key $\mathbf{s_1}$ to attackers. This method is applicable to both deterministic and randomized versions of Dilithium.

- **Method 2:** We generate a faulty challenge c^* and then utilize the DFA method of [BP18] to recover $\mathbf{s_1}$. The attacker let Alice sign the same message twice and leverage the difference between the correct and fualty signatures to recover the private key. This method is only applicable to the deterministic version of Dilithium.

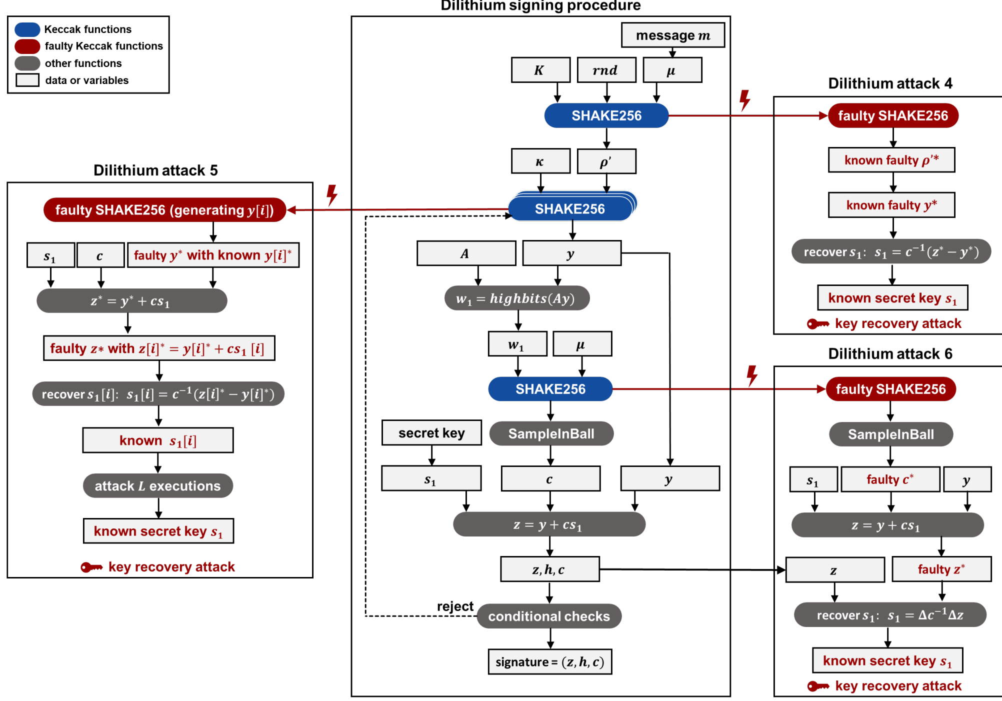


Figure 18: Dilithium signing procedure and our attacks

Following Method 1, we propose two attacks.

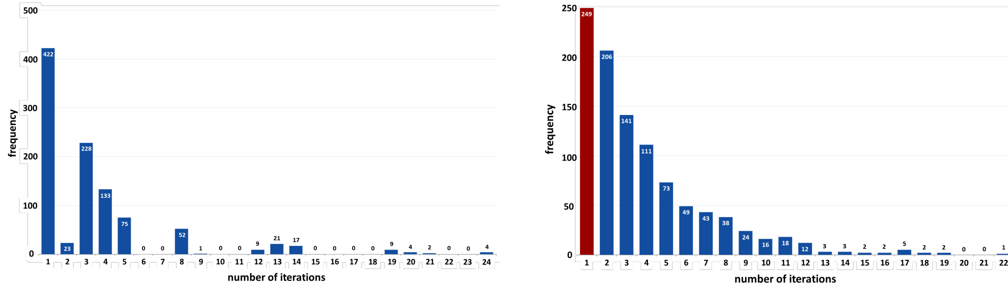
- **Dilithium Attack 4 (control-rhoprime):** As shown in Figure 18, we set the output of the SHAKE256 instance that generates ρ' to a known value. Using the known ρ', we can further calculate $\mathbf{y}^*$ corresponding to each κ. Finally, the secret key $\mathbf{s_1}$ can be solved from the faulty signature $\sigma^* = (\mathbf{z}^*, \mathbf{h}^*, c^*)$ by $\mathbf{s_1} = c^{*-1}(\mathbf{z}^* - \mathbf{y}^*)$.

  The Dilithium signing procedure follows the Fiat-Shamir with Abort scheme. Since the attacker does not know the private key, they cannot determine the exact number of iterations executed. However, this attack targets the generation of ρ', which is carried out outside the abort loop, so $\mathbf{y}$ for each round is generated using the same faulty ρ'. Therefore, the attacker can guess the number of iterations and calculate the corresponding $\mathbf{y}$ using κ, allowing them to recover the private key. As shown in Figure 19(a), we simulated the case where ρ' is set to zero due to a fault attack. The x-axis represents the number of iterations of the abort loop, and the y-axis represents the frequency of each occurrence. We conducted 1000 experiments and observed that the number of iterations does not exceed 24. The attacker can recover the correct key within a few guesses (78.3% success rate within 3 guesses).

- **Dilithium Attack 5 (control-y-coefficients):** $\mathbf{y}$ is an array of L (a Dilithium parameter) polynomial elements. The process of generating each polynomial involves

a SHAKE256 instance. As shown in Figure 18, we attack one of the SHAKE256 instances each execution, making the output polynomial become a known value. For example, by faulting the SHAKE256 instance for the i-th polynomial, we change $\mathbf{y}$ from $\mathbf{y} = [y_1, y_2, \ldots, y_L]$ to $\mathbf{y}^* = [y_1, y_2, \ldots, y_i^*, \ldots, y_L]$. With the faulty $\mathbf{y}^*$, we can recover the value of the corresponding polynomial element in $\mathbf{s_1}$ by $\mathbf{s_1}[i] = c^{*-1} \cdot (\mathbf{z}^*[i] - \mathbf{y}^*[i])$. By sequentially attacking L polynomials of $\mathbf{y}$ (where each $\mathbf{y}$ can be different), we can recover the complete $\mathbf{s_1}$, enabling us to forge a signature.

For this attack, since the target is located within the Fiat-Shamir with Abort loop, the attack can only succeed if the faulty round passes the check. Otherwise, the re-generated $\mathbf{y}$ will be the non-faulty value. We simulate the fault injection scenario by zeroizing out one of the polynomials of $\mathbf{y}$, signing with random private keys and messages for 1000 times, As shown in Figure 19(b), the attack is successful only when the number of iterations is 1 (the first column), and the success rate is 24.9%.



(a) No. iterations for Dilithium Attack 4

(b) No. iterations for Dilithium Attack 5

Figure 19: No. iterations of the Fiat-Shamir with Abort loop

Based on method 2, we propose our sixth fault attack against Dilithium.

- **Dilithium Attack 6 (control-c-DFA):** As shown in Figure 18, using the Keccak fault attack we proposed, a different faulty challenge c^* is generated. Given the correct $\mathbf{z} = \mathbf{y} + c \cdot \mathbf{s_1}$ and faulty $\mathbf{z}^* = \mathbf{y} + c^* \cdot \mathbf{s_1}$, we can calculate the difference and recover the key $\mathbf{s_1}$ by $\mathbf{s_1} = \Delta c^{-1} \cdot \mathbf{\Delta z}$, where $\Delta c = c^* - c$ and $\Delta \mathbf{z} = \mathbf{z}^* - \mathbf{z}$.

3.4.3 Attacking the Verification Procedure of Dilithium

According to Algorithm 7, the implementation of the Dilithium verification procedure can be represented by Figure 21. It starts by computing μ, the hash value of tr and m. Then, the $\mathbf{w_1'}$ is calculated using the signature and the public key. After that, μ and $\mathbf{w_1'}$ are hashed by a SHAKE256, generating c'. Finally, if c' is equal to c and the checks of $\mathbf{z}$ and $\mathbf{h}$ are satisfied, the verification is successful.

Figure 20 illustrates our attack method during the verification procedure. By injecting faults, we make the verification process accept an incorrect signature corresponding to any message chosen by the attacker.

- **Dilithium Attack 7 (control-cprime-bypass):** As shown in Figure 21, we set the output of the SHAKE256 generating c' known fixed value, donated as c^*. Therefore, as long as c' equals c^*, and $\mathbf{z}$ and $\mathbf{h}$ are valid, the verification can be bypassed.

  Specifically, the attack can be carried out in the following steps: Firstly, the attacker generates $\mathbf{z}^*$ and $\mathbf{h}^*$ that can pass the condition checks (e.g., zero vectors). Secondly, the attacker sets the value of c in the signature to be equal to c^*, where c^*, is one of

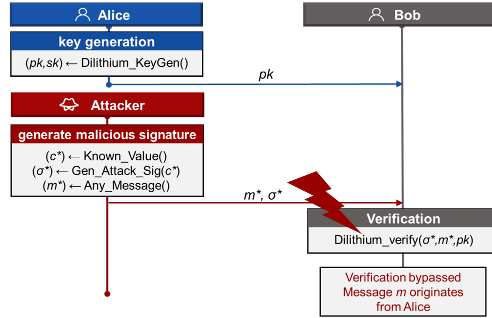


Figure 20: Attack method during the verification procedure

the candidate values that our Keccak fault attacks can generate. The attacker then inputs the incorrect signature and **any** message m into the verification algorithm. By injecting a fault during the execution, the value c' is replaced with c^*, passing all the verifications and achieving a verification bypass attack.

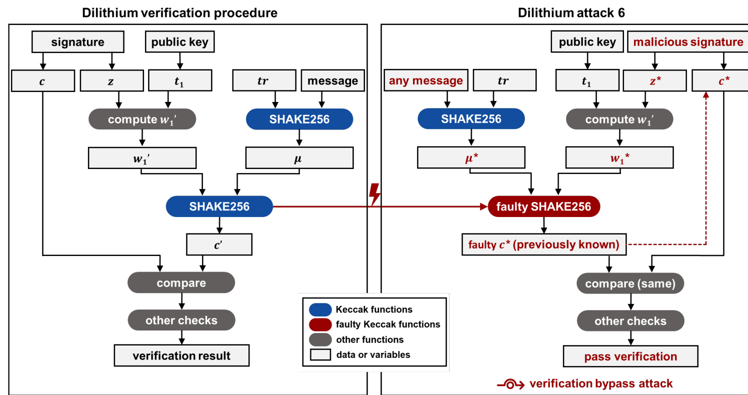


Figure 21: Dilithium verification procedure and our attacks

4 Experiments

Note that this section is one of the most important sections of this paper. In Section 4.2, we perform fault characterization on five ARM-based MCUs, covering the widely-used single-core and multi-core ARM architectures like Cortex-M0+, Cortex-M3, Cortex-M4, and Cortex-M33, demonstrating the feasibility of the loop-abort faults. More importantly, we provide a guide for reliably inducing loop-abort faults on ARM Cortex-M MCUs in a short time. Then, Section 4.3 explains the causes of loop-abort faults through experiments. Section 4.4 introduces our fault trigger techniques based on binary rewriting and side-channel analysis. Finally, in section 4.5, we validate our practical attacks on real-world Kyber and Dilithium implementations. Our experimental code is open-source.

4.1 Experimental Setup

Our experiments were conducted on the official implementations of Kyber and Dilithium [pcb, pca]. To demonstrate the feasibility of the loop-abort attack on ARM Cortex-

M MCUs, we selected five MCUs as the Device Under Test (DUT): the Cortex-M0+ STM32L073RZT6U, the Cortex-M3 STM32F103RCT6, the Cortex-M4 STM32F407ZGT6, the Cortex-M4 STM32F405RGT6, and the Cortex-M33 dual core NXP LPC55S69JBD100. These DUTs cover the four most commonly used ARM architectures, including both single-core and multi-core processors.

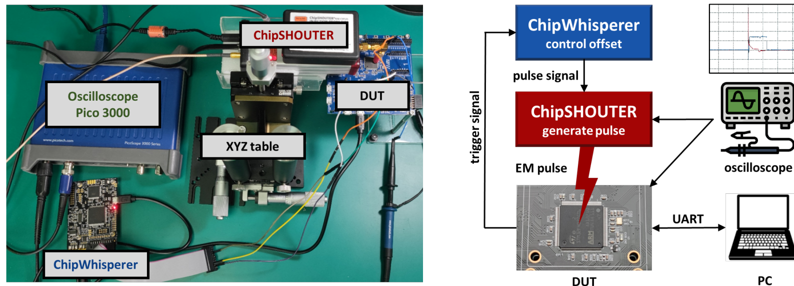


Figure 22: Experimental Setup

We use EMFI to implement our attacks, and the experimental setup is shown in Figure 22. We communicate with the DUT using a UART connection, and the DUT executes the target program upon receiving the command. When the execution reaches the target code segment, the DUT activates a trigger signal. We use a NewAE ChipWhisperer [Newa] to precisely control the offset after receiving the trigger signal from the DUT, and then issue a pulse signal to instruct the NewAE ChipSHOUTER [Newb] electromagnetic pulse generator to perform the fault injection. The ChipWhisperer is also responsible for rebooting the DUT when a fault occurs. The ChipSHOUTER is mounted on an XYZ table to enable fine-grained spatial positioning adjustments. Finally, we collect the output information through the UART for further analysis. We also use a PicoScope 3000 series oscilloscope [pT] to observe the trigger signal and the electromagnetic pulse signal, which assists in the localization of the fault injection. Our experimental setup implements automated fault injection and result collection.

4.2 Fault characterization

We characterized the five DUTs in the spatial, time, and pulse intensity dimensions and determine the optimal fault injection parameters. We provided a method that leverages fault characteristics to reliably induce loop-abort faults quickly. We adopted a simple iterative array assignment as the target code for our characterization. This code is part of the Keccak implementation in Kyber and Dilithium.

4.2.1 Fault Characterization in Space Dimension

We divided the MCU chip packages into grid cells with side lengths of 0.5 mm. For each grid cell, we adjust the parameters and conduct 1000 fault injection attempts, measuring the fault rate and the corresponding optimal pulse intensity. Figure 23-27 show the experimental results on five DUTs. The column on the left indicates the fault occurrence rate at different spatial locations. The red cells represent the locations where the target fault may occur, while the gray cells represent the locations where only crash or reset faults were observed. The column on the right shows the corresponding optimal pulse intensity, with the values representing the voltage of the ChipSHOUTER electromagnetic coil. One can select the optimal fault space location and corresponding pulse intensity based on these results to maximize the fault success rate.

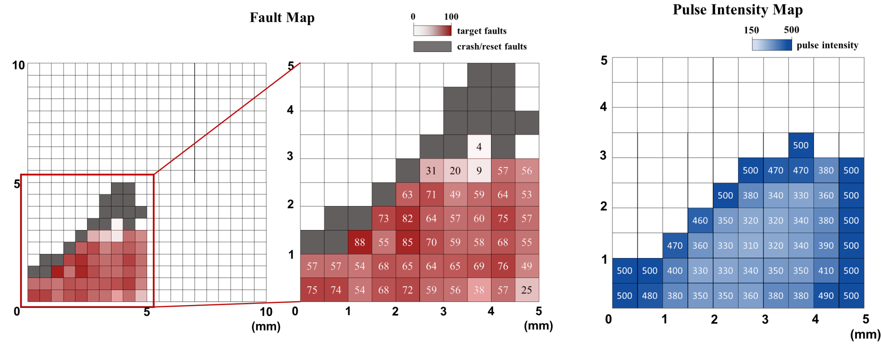


Figure 23: Spatial characterization results of STM32L073RZT6U

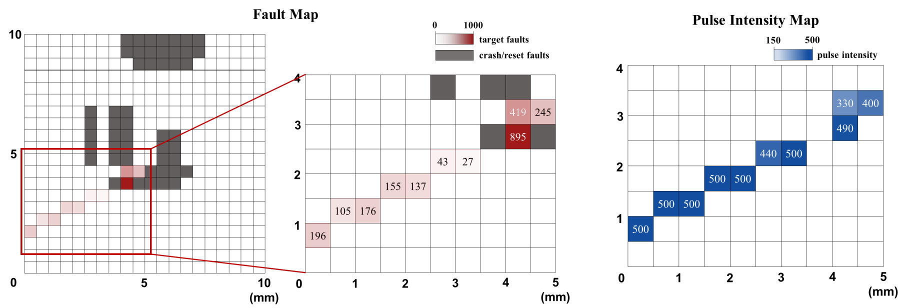


Figure 24: Spatial characterization results of STM32F103RCT6

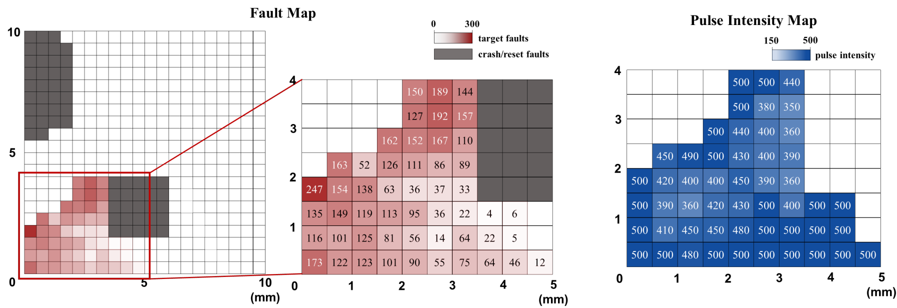


Figure 25: Spatial characterization results of STM32F405RGT6

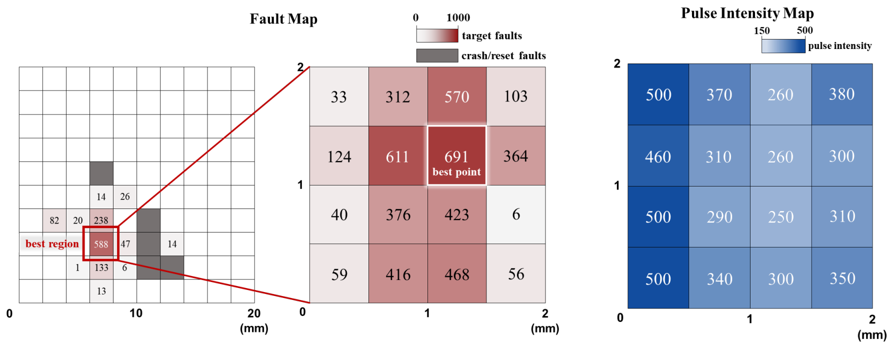


Figure 26: Spatial characterization results of STM32F407ZGT6

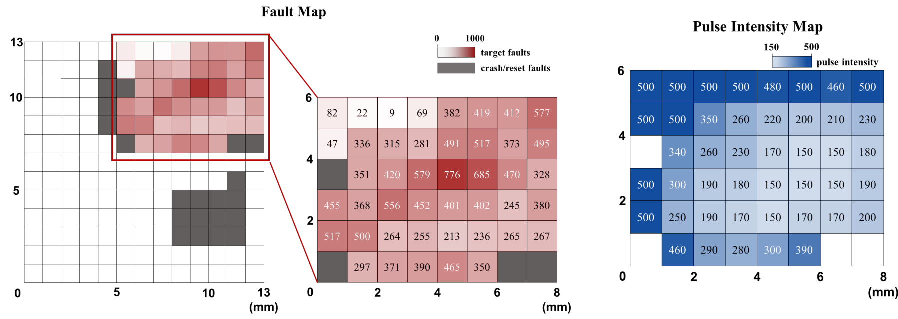
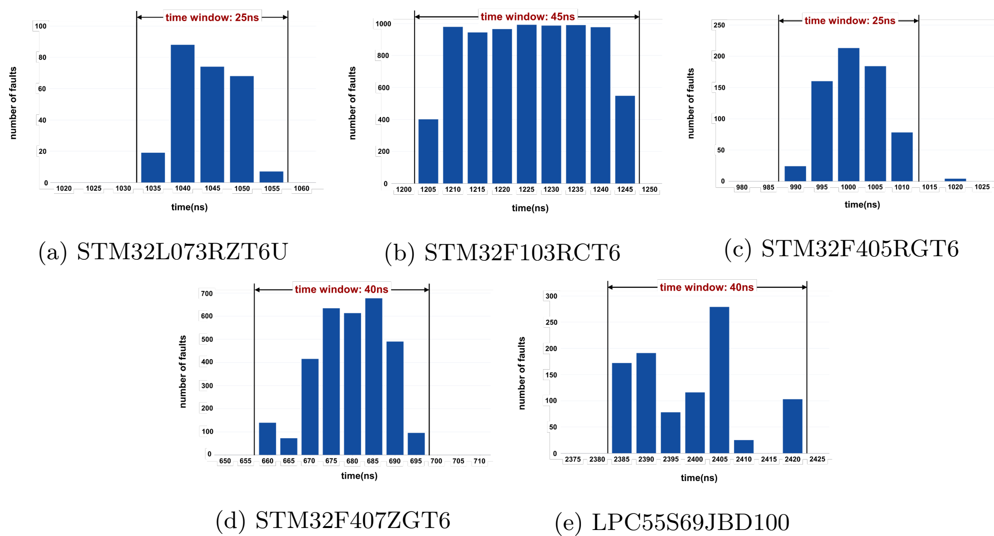


Figure 27: Spatial characterization results of LPC55S69JBD100

We summarize the following spatial fault characteristics of Cortex-M MCUs:

- The spatial distribution of faults tends to be concentrated rather than dispersed. Although the shapes of fault-inducing regions can vary across different MCUs, these regions are often large and have continuous shapes, such as rectangles or trapezoids. Therefore, attackers do not need highly precise spatial targeting to inject loop-abort faults successfully.
- The fault rate is not necessarily higher in locations closer to the distribution center, as these areas are more prone to crash faults.
- The central part of the fault distribution region requires a smaller pulse intensity, while the edges require a larger one. This indicates that the affected electronic components are located in the middle of the region.

We do not characterize at a higher precision, such as 0.1 mm. First, the faults are not highly sensitive to the probe's position. Second, we use a fault injection probe with a ferrite core diameter of 1 mm and a coil diameter of approximately 2 mm, and the electromagnetic field changes can have a broader impact.



(a) STM32L073RZT6U (b) STM32F103RCT6 (c) STM32F405RGT6

(d) STM32F407ZGT6 (e) LPC55S69JBD100

Figure 28: Time characterization of the DUTs

4.2.2 Fault Characterization in Time Dimension

We set the core frequency of all five DUTs to 16 MHz, select appropriate spatial positions and pulse intensities, and measure the fault rate of the DUTs with injected faults at different time offsets. We use the Chipwhisperer's 200 MHz clock to achieve a precise time delay with 5 ns accuracy. For each delay, we perform 1000 fault injection attempts. Figure 28 shows our results.

The results show that the Cortex-M MCU exhibits **an unimodal distribution** of fault rates over time, with a 25-45 ns time window, which is in the same order of magnitude as the 62.5 ns duration of a single instruction at a 16 MHz clock frequency. The faults are highly time-sensitive.

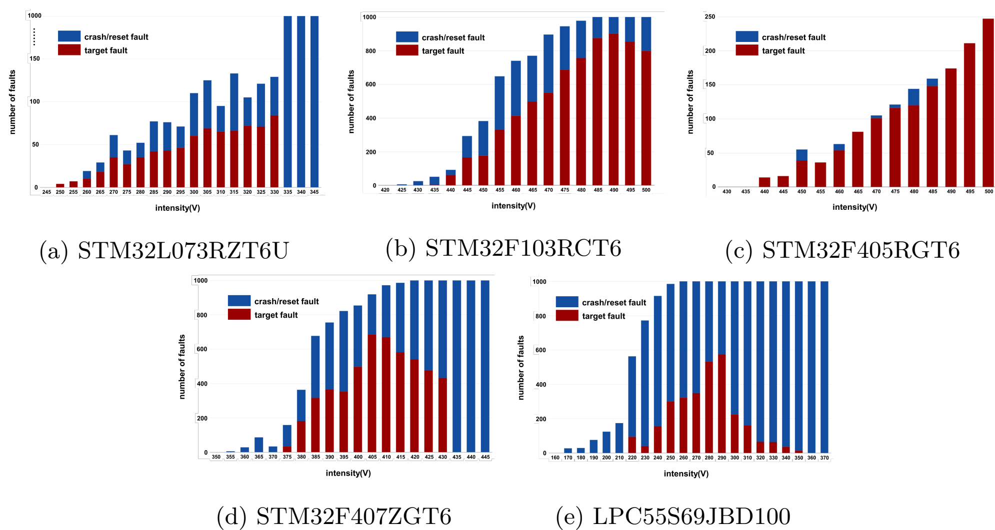
4.2.3 Fault Characterization in the Pulse Intensity Dimension

We fix the spatial position and offset and measure the relationship between faults and pulse intensity. The electromagnetic pulse generator ChipSHOUTER can change the intensity by setting the coil voltage value, which ranges from 150 V to 500 V. We select values around the optimal intensity and perform 1000 fault injection attempts at each intensity value. The results are shown in Figure 29.

We summarize the following pulse intense fault characteristics of Cortex-M MCUs:

- The overall fault rate increases with increasing pulse intensity, and no faults will occur when the pulse intensity is too low.
- Both target fault and crash/reset fault can be observed simultaneously. When the intensity exceeds a certain threshold, a crash/reset is 100% probable. For example, the STM32L073 DUT will crash/reset when the intensity is greater than 335 V.
- Loop-abort fault rate increases as the pulse intensity increases when it is relatively small, reaching a peak and gradually decreasing. This is because strong pulses can trigger more crashes/resets. However, on the STM32F405 DUT, the targeted fault rate increases without reaching a peak, indicating that the optimal intensity exceeds the maximum intensity we can provide.

The pulse intensity should not be too large or too small when inducing a loop-abort fault. Otherwise, there will be a majority of crashes/resets or no faults.



(a) STM32L073RZT6U (b) STM32F103RCT6 (c) STM32F405RGT6

(d) STM32F407ZGT6 (e) LPC55S69JBD100

Figure 29: Pulse intensity characterization of the DUTs

4.2.4 A Guide of Loop-Abort Fault Injection

Based on the characteristics of fault rate over time, space, and pulse intensity, we propose the following heuristic method to quickly determine the parameters and stably induce loop-abort fault on ARM Cortex-M devices:

- **Step1: Fast fault region localization based on hardware spatial similarity.** Adjust the pulse intensity to the maximum and scan the DUT spatially. Because fault regions are often large and have continuous shapes, one can just characterize the edge of it to determine the location. Since the fault rate increases with pulse intensity, one can determine all the areas where faults may occur. This step has not yet performed the time-domain localization, so the results obtained are often crash faults.
- **Step2: Determine the time offset.** Select the location of the fault region center, reduce the pulse intensity, and perform a fine-grained time-domain scan. If loop-abort faults occur, determine the best time offset; otherwise, re-select the location. The scanning step should be smaller than the time window (25-45 ns).
- **Step3: Precise parameter localization based on the trisection method.** With the offset fixed, adjust the position within the fault region and determine the best pulse intensity for each point. Since the fault rate distribution is an unimodal function of the pulse intensity, the peak value can be quickly determined using the trisection method: **(1)** Define the interval $[l, r]$ where l and r are the lower and upper bounds of the pulse intensity. **(2)** Divide the interval into three parts by calculating two midpoints $m_1 = l + (r - l)/3$ and $m_2 = r - (r - l)/3$. Then measure the fault rates at m_1 and m_2: FR_1 and FR_2. If $FR_1 < FR_2$, the maximum must be in the interval $[m_1, r]$. Set $l = m_1$. Otherwise, set $r = m_2$. **(3)** Repeat (2) until the interval is sufficiently small. The best point is $(l + r)/2$.

4.3 Fault Explanation

We use the code snippet in Figure 30 as an example to explain the reason for the fault. It is a simple iterative array assignment operation. In the assembly code, a register, such as $r0$, stores the loop counter i. First, the initial value 0 is written to this register, and then a jump is made to the comparison instruction, which serves as the loop exit condition. If the condition is satisfied, the loop exits; otherwise, a jump is made, and the loop body is executed.

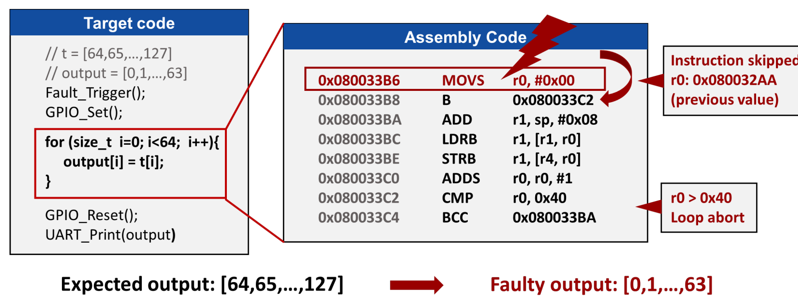


Figure 30: Fault explanation of our loop-abort attacks

We design the following experiment: First, we assign an initial value to the output array at the beginning of the program. Then, we inject faults during the execution of the iterative assignment code and observe the output values. The results indicate that the

target array is still the initial value we specified. This suggests that the faults did not directly alter the array's values but rather skipped the entire assignment loop.

The signals observed with the oscilloscope also prove the occurrence of loop-abort faults. In Figure 31, the red signal represents our electromagnetic pulse, while the blue one represents the GPIO signal, which is set before and reset after the loop. The GPIO reset is significantly earlier when faults are induced, indicating loop-abort faults.

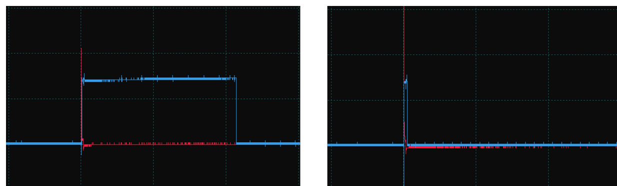


Figure 31: Normal signal (left) and faulty signal (right)

We further analyze how the loop-abort fault is achieved. We output the value of the loop counter after the iterative assignments. As shown in Figure 30, when a valid fault occurs, the loop counter is also abnormal and tends to be random. This suggests that the previous value is still stored in the register, and the cause of the fault is skipping the **MOV** instruction. If the register value directly satisfies the loop termination condition, the loop body will not be executed, thus achieving the loop-abort fault. In addition, if the branch instruction is also skipped, the program will enter the loop body with a faulty register value, leading to a crash.

Another cause of the fault is skipping both assignment instruction and branch instruction, resulting in the loop executing only once. For example, This fault occurs frequently when we inject faults during the Keccak-f permutation. We output the value of the loop counter and find out that the loop counter has a faulty value. Therefore, we explain the cause of the fault as skipping the **STR** and **B** instructions. The loop counter is stored in the stack, so skipping the **STR** instruction is equivalent to skipping the initialization of the counter. Skipping the **B** instruction is equivalent to skipping the comparison and directly entering the loop body. We added a statement inside the loop body to set the counter to a larger value, simulating the effect of our fault model. We obtained the same output as the faulty output, thus validating our theory.

4.4 Fault Triggering

In our practical evaluation of the attack, we insert code at the beginning of the target loop to generate a trigger signal for locating the target. Using binary rewriting, we can obtain an altered implementation with the added trigger. Additionally, we can determine the time window for fault injection with the assistance of electromagnetic or power side-channel information.

4.4.1 Trigger Insertion Based on Binary Rewriting

Qu et al. proposed a binary instrumentation framework called PIFER for bare-metal embedded firmware and open-sourced a prototype that supports all ARM Cortex-M microprocessors [QZZG23]. With this framework, as long as the attacker can obtain the binary of the implementation of Kyber or Dilithium, they can insert fault triggers before the target Keccak instance or the target loop, improving the success rate of fault injection.

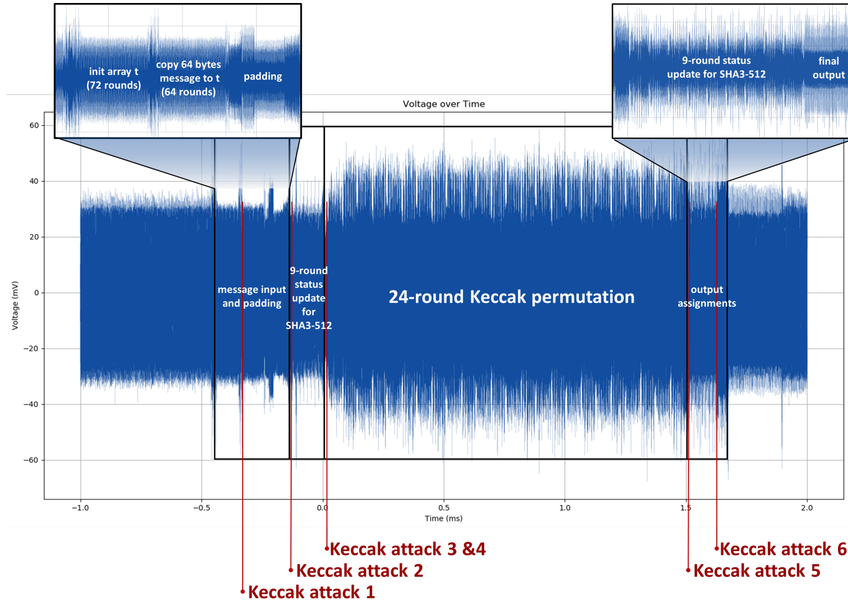


Figure 32: Side-channel trace of Keccak on ARM Cortex-M MCU

4.4.2 Side-channel Assisted Fault Triggering

We used a near-field EM probe to measure the electromagnetic side-channel traces of a SHA3-512 instance of Kyber executing on an STM32F407ZGT6 DUT. The measurement results are shown in Figure 32, where the characteristic 24-round permutation structure of Keccak can be clearly observed. Attackers can leverage this feature to locate Keccak in the side-channel traces of Kyber and Dilithium. We labeled each part of the trace in the figure, indicating the corresponding operation or computation. We also identified the six Keccak attack injection points we proposed. This information lets the attacker determine the offsets and precisely locate the fault injection timing.

4.5 Fault Injection Results

We validate the practicality of our proposed attacks on Keccak, Kyber, and Dilithium on the STM32F407ZGT6 DUT. Our results demonstrate that our attack methods have a high success rate and are applicable to all procedures of Kyber and Dilithium. Based on fault characterizing results, we set the spatial location of fault injection to the best point marked in Figure 26. We adjust the best time offset and pulse intensity determined in our characterizing to find the optimal parameters for each attack.

4.5.1 Practical Attacks Against Keccak

We target the SHA3-512 function in Kyber and Dilithium as the attack objective and validate the six Keccak attacks we propose. We use a 64-byte input for the function, simulating Keccak instances in Kyber and Dilithium key generation. We insert a trigger in front of the target loop and measure the success rate and optimal attack parameters for each attack. We performed 1000 fault injection attempts for each fault parameter setting and the results are shown in Table 7. Our six attacks successfully obtained the expected faulty outputs, with success rates ranging from 12.4% to 65.1%. The best time offset and pulse intensity of the six attacks are not significantly different, which proves that our loop-abort faults are highly reproducible.

Table 7: Fault rate of our Keccak attacks

Keccak Attack	Fault Rate	Time Offset (ns)	Pulse Intensity (V)
Keccak Attack 1	56.7%	665	260
Keccak Attack 2	20.9%	665	260
Keccak Attack 3	12.4%	660	290
Keccak Attack 4	49.8%	680	250
Keccak Attack 5	36.3%	685	260
Keccak Attack 6	65.1%	680	260

Note that in our attack scheme, successfully conducting a Keccak attack and controlling the Keccak output is approximately equivalent to successfully executing our attack strategy against Kyber and Dilithium. To achieve our higher-level attack strategy, the attacker only needs to target different Keccak instances in Kyber and Dilithium. For the completeness of the practical attacks, we provide experimental validation of complete attacks below. Several auxiliary solver tools for our attack scheme are provided in our open-source code.

4.5.2 Practical Attacks on Kyber

We targeted the official implementation of Kyber768 on the STM32F407ZGT6 DUT. We validated one attack for each of the key generation, encapsulation, and decapsulation procedures of Kyber768, demonstrating that our attacks are applicable to all procedures. Additionally, we validated Kyber Attack 2 as a representative of a multi-point fault attack, proving the feasibility of the proposed attacks. In our experiments, we used the Keccak Attack 1 as the primary attack to implement the Kyber fault attacks.

Attacking the key generation procedure: We verified the Kyber Attack 1, which targets the SHA3-512 instance that generates the random seed. We conducted 1000 fault injection attempts, and under a 680 ns time offset and a 260 V pulse intensity, we achieved 551 loop-abort faults. We observed that the secret key generated, which should have been random, became fixed values when the fault occurred. We calculated the final private key based on the known Keccak intermediate state, which matched the observed results. Therefore, we can recover the private key with a 100% probability when the fault occurs, and subsequently decrypt the ciphertext to obtain the shared secret.

Attacking the encapsulation procedure: We verified the Kyber Attack 6, which targets the SHA3-512 used to generate the random seed coins for the IND-CPA encryption. We performed 1000 fault injection attempts, and at a time offset of 685 ns and a pulse intensity of 250 V, we obtained 476 successful loop-abort faults. Our experiments used different random key pairs, and upon fault occurrence, we can recover the *coins* and the corresponding shared secret with a 100% probability.

Attacking the decapsulation procedure: We verified the Kyber Attack 8, which targets the SHAKE256 instance generating the shared secret. We performed 1000 fault injection attempts, and at a time offset of 680 ns and a pulse intensity of 260 V, we obtained 405 successful loop-abort faults. When the fault occurs, the value of the shared secret is fixed, and we can recover it directly with a 100% probability.

Attacking using our multi-point fault attack: We verified the Kyber Attack 2, which targets the SHAKE256 instances used to generate the private key. For Kyber768, we need to successfully inject three faults in a single execution to recover the private key. We first tested the success rate of attacking a single SHAKE256 instance, performing 1000 fault injection attempts and obtaining 424 successful loop-abort faults. We further tested the success rate of the complete attack, with 27 successes out of 1000 attempts. This demonstrates that our multi-point fault injection attack is feasible. However, given their relatively low success rates, we recommend using our proposed single-point FIA methods.

4.5.3 Practical Attacks on Dilithium

We targeted the official implementation of Dilithium2 and performed fault injection experiments on the STM32F407ZGT6 DUT. We validated one attack for each of the key generation, signing, and verification procedures of Kyber768, demonstrating that our attacks are applicable to all of them.

Attacking the key generation procedure: We verified the Dilithium attack 1, which targets the SHAKE256 instance that generates the random seed. We conducted 1000 fault injection attempts, and with a 705 ns time offset and a 240 V pulse intensity, we successfully achieved 293 loop-abort faults. When the fault occurs, we can recover the private key with a 100% probability and subsequently forge signatures.

Attacking the signing procedure: We verified the Dilithium attack 4, which targets the SHAKE256 instance used in the generation of y. We conducted 1000 fault injection attempts, and with a 690 ns time offset and a 260 V pulse intensity, we successfully achieved 251 loop-abort faults. The attack also requires that c is reversible, which occurred in 100% of our 1000 tests. The probability of a polynomial being reversible in R_q is $(1 - 1/q)^n$ [LPR13], which for Dilithium is approximately $1 - 2^{-15}$ [BP18]. We then recovered ρ' with 100% probability. Based on the distribution shown in Figure 19(b), we guessed the κ and recover the secret key with a probability of 79.0% within three attempts.

Attacking the verification procedure: We verified the Dilithium attack 7, which targets the SHAKE256 instance that generates c^* during verification. We conducted 1000 fault injection attempts, and with a 690 ns time offset and a 270 V pulse intensity, we successfully achieved 505 loop-abort faults. We generated malicious signatures and random messages, where c is set to the pre-known value c^*. Upon successful fault injection, we can make the faulty signature pass the verification with a 100% probability.

5 Countermeasures

In this section, we discuss countermeasures against our proposed fault injection attacks. Package-on-package, electromagnetic shielding, and sensor-based solutions can prevent electromagnetic fault injection [HMNS18, EBRM16]. However, requiring all Kyber and Dilithium implementations to be deployed on devices with such measures is impractical. Therefore, we need to consider software-level countermeasures.

5.1 Implementation-level Countermeasures

These countermeasures strengthen the security of existing Keccak implementations, making loop-abort attacks more difficult or even infeasible to perform.

- **Reducing the Use of Keccak:** One of the main contributions of our work is to point out the excessive use of Keccak to generate or compute secret information in Kyber and Dilithium, leading to severe FIA vulnerabilities. Therefore, the most effective countermeasure is to reduce the use of Keccak and, if possible, use a hardware random number generator to generate random numbers directly. Additionally, the output of a Physical Unclonable Function (PUF) can be used as a source of randomness, achieving inherent security [OSD04, Kay20].

- **Jitter implementation:** Our loop-abort fault has a time window of only 25-45 ns when running on a 16 MHz device, requiring precise fault injecting. Therefore, we can introduce jitter countermeasures, such as randomly inserting NOP instructions in the implementation. Note that this measure cannot completely prevent our attack, as a patient attacker can still use a large number of attempts to collide the time window, and as long as they succeed once, the attack can still be successful. The advantage of this measure is that it is performance-friendly and easy to implement.

- **Loop unrolling:** Unrolling the loops can fundamentally prevent our attack. However, the code size will significantly increase after unrolling, making it difficult to deploy on resource-constrained embedded devices.

5.2 Verification-based Countermeasures

These countermeasures introduce additional checks into the original algorithm to ensure it is executed correctly.

- **Blacklist Checking:** Many of our Keccak attacks set the Keccak output to become a fixed value, so an effective countermeasure is to add these values to a blacklist. This countermeasure checks whether the Keccak output or intermediate states belong to the blacklist. To improve efficiency, one can sample a few random bits from the output and compare them against the blacklist to detect attacks.
- **Execution Time Checking:** Executions with loop-abort fault injected take less time than normal execution. We can add timing checkpoints before and after loops to verify that each loop execution is as expected. Considering the performance of the algorithm, we recommend using this countermeasure only in important loops.
- **Redundancy Checking:** Repeatedly execute Keccak instances in Kyber and Dilithium and compare the results to ensure they are correct. Note that this countermeasure only increases the difficulty of fault attacks. Attackers can still attack multiple Keccak instances in a single execution, which our experiments have demonstrated is feasible. Additionally, this countermeasure incurs significant overhead.

6 Conclusion and Future Work

In this paper, we demonstrated that the vulnerability of Keccak under loop-abort faults enables numerous FIAs on Kyber and Dilithium implementations. By performing fault characterizing on five different devices belonging to four different series, we have shown that loop-abort fault attacks are prevalent on ARM Cortex-M MCUs, achieving success rates ranging from 8.8% to 89.5%. We also demonstrated that our proposed attacks apply to all procedures of the official implementations of Kyber and Dilithium. When a fault is successfully injected, it allows for the recovery of private keys, shared secrets, and other sensitive information, as well as the forgery of signatures and bypassing verification with 100% probability. Our work indicates that the excessive use of Keccak in generating and computing secret information leads to severe vulnerabilities of Kyber and Dilithium, highlighting the need for fault protection countermeasures for Keccak and security reinforcement for Kyber and Dilithium.

Despite the high generality and practicality of our attack scheme, which poses a severe threat to Kyber and Dilithium, there are some limitations that will be the focus of our future work. First, although our work provides multiple optional injection points, attackers still need extensive experiments to precisely determine the accurate timing for fault injection. We plan to explore more possibilities for fault attacks on Keccak and attempt to propose improved attack schemes against Kyber and Dilithium with more relaxed timing assumptions. Second, our proposed attacks often render the Keccak output to fixed values, making them relatively easy to detect. During our fault characterization, we also observed that injecting faults at the right timing can skip the assignment of a single array element. Hence, we plan to utilize this capability to construct more covert FIAs against Kyber and Dilithium. Additionally, we plan to explore the potential of leveraging the Keccak vulnerabilities to compromise the security of other lattice-based post-quantum cryptographic algorithms, such as Falcon [FHK+18], BIKE [ABB+22], and HQC [MAB+18].

References

[ABB+22] Nicolas Aragon, Paulo Barreto, Slim Bettaieb, Loïc Bidoux, Olivier Blazy, Jean-Christophe Deneuville, Philippe Gaborit, Santosh Ghosh, Shay Gueron, Tim Güneysu, et al. Bike: bit flipping key encapsulation. 2022.

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

[BDK+18] Joppe Bos, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, John M Schanck, Peter Schwabe, Gregor Seiler, and Damien Stehlé. Crystals-kyber: a cca-secure module-lattice-based kem. In *2018 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 353–367. IEEE, 2018.

[BDPVA07] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. Sponge functions. In *ECRYPT hash workshop*, volume 2007, 2007.

[BDPVA13] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. Keccak. In *Annual international conference on the theory and applications of cryptographic techniques*, pages 313–314. Springer, 2013.

[BGS15] Nasour Bagheri, Navid Ghaedi, and Somitra Kumar Sanadhya. Differential fault analysis of sha-3. In *Progress in Cryptology–INDOCRYPT 2015: 16th International Conference on Cryptology in India, Bangalore, India, December 6-9, 2015, Proceedings 16*, pages 253–269. Springer, 2015.

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

[chr] Chromium blog. https://blog.chromium.org/2023/08/protecting-chrome-traffic-with-hybrid.html. Accessed: 2024-07-07.

[DDP21] Jeroen Delvaux and Santos Merino Del Pozo. Roulette: Breaking kyber with diverse fault injection setups. *IACR Cryptol. ePrint Arch.*, 2021:1622, 2021.

[DKL+18] Léo Ducas, Eike Kiltz, Tancrede Lepoint, Vadim Lyubashevsky, Peter Schwabe, Gregor Seiler, and Damien Stehlé. Crystals-dilithium: A lattice-based digital signature scheme. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 238–268, 2018.

[DTVV19] Jan-Pieter D'Anvers, Marcel Tiepelt, Frederik Vercauteren, and Ingrid Verbauwhede. Timing attacks on error correcting codes in post-quantum schemes. In *Proceedings of ACM Workshop on Theory of Implementation Security Workshop*, pages 2–9, 2019.

[EBRM16] David El-Baze, Jean-Baptiste Rigaud, and Philippe Maurine. A fully-digital em pulse detector. In *2016 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 439–444. IEEE, 2016.

[EFGT17] Thomas Espitau, Pierre-Alain Fouque, Benoît Gérard, and Mehdi Tibouchi. Loop-abort faults on lattice-based fiat-shamir and hash-and-sign signatures. In *Selected Areas in Cryptography–SAC 2016: 23rd International Conference, St. John's, NL, Canada, August 10-12, 2016, Revised Selected Papers 23*, pages 140–158. Springer, 2017.

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

[ES03] Niklas Eén and Niklas Sörensson. An extensible sat-solver. In *International conference on theory and applications of satisfiability testing*, pages 502–518. Springer, 2003.

[FHK+18] Pierre-Alain Fouque, Jeffrey Hoffstein, Paul Kirchner, Vadim Lyubashevsky, Thomas Pornin, Thomas Prest, Thomas Ricosset, Gregor Seiler, William Whyte, Zhenfei Zhang, et al. Falcon: Fast-fourier lattice-based compact signatures over ntru. *Submission to the NISTs post-quantum cryptography standardization process*, 36(5):1–75, 2018.

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

[HMNS18] Th Heckmann, Konstantinos Markantonakis, David Naccache, and Th Souvignet. Forensic smartphone analysis using adhesives: Transplantation of package on package components. *Digital Investigation*, 26:29–39, 2018.

[HPP21] Julius Hermelink, Peter Pessl, and Thomas Pöppelmann. Fault-enabled chosen-ciphertext attacks on kyber. In *Progress in Cryptology–INDOCRYPT 2021: 22nd International Conference on Cryptology in India, Jaipur, India, December 12–15, 2021, Proceedings 22*, pages 311–334. Springer, 2021.

[IMS+22] Saad Islam, Koksal Mus, Richa Singh, Patrick Schaumont, and Berk Sunar. Signature correction attack on dilithium signature scheme. In *2022 IEEE 7th European Symposium on Security and Privacy (EuroS&P)*, pages 647–663. IEEE, 2022.

[Kay20] Turgay Kaya. A true random number generator based on a chua and ro-puf: design, implementation and statistical analysis. *Analog Integrated Circuits and Signal Processing*, 102:415–426, 2020.

[KDK+14] Yoongu Kim, Ross Daly, Jeremie Kim, Chris Fallin, Ji Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. Flipping bits in memory without accessing them: An experimental study of dram disturbance errors. *ACM SIGARCH Computer Architecture News*, 42(3):361–372, 2014.

[KLS18] Eike Kiltz, Vadim Lyubashevsky, and Christian Schaffner. A concrete treatment of fiat-shamir signatures in the quantum random-oracle model. In *Advances in Cryptology–EUROCRYPT 2018: 37th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Tel Aviv, Israel, April 29-May 3, 2018 Proceedings, Part III 37*, pages 552–586. Springer, 2018.

[KPLG24] Elisabeth Krahmer, Peter Pessl, Georg Land, and Tim Güneysu. Correction fault attacks on randomized crystals-dilithium. *Cryptology ePrint Archive*, 2024.

[LAFW18] Pei Luo, Konstantinos Athanasiou, Yunsi Fei, and Thomas Wahl. Algebraic fault analysis of sha-3 under relaxed fault models. *IEEE Transactions on Information Forensics and Security*, 13(7):1752–1761, 2018.

[LFZD16] Pei Luo, Yunsi Fei, Liwei Zhang, and A Adam Ding. Differential fault analysis of sha3-224 and sha3-256. In *2016 Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*, pages 4–15. IEEE, 2016.

[LPR13] Vadim Lyubashevsky, Chris Peikert, and Oded Regev. A toolkit for ring-lwe cryptography. In *Annual international conference on the theory and applications of cryptographic techniques*, pages 35–54. Springer, 2013.

[LS15] Adeline Langlois and Damien Stehlé. Worst-case to average-case reductions for module lattices. *Designs, Codes and Cryptography*, 75(3):565–599, 2015.

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

[Lyu12] Vadim Lyubashevsky. Lattice signatures without trapdoors. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 738–755. Springer, 2012.

[MAB+18] Carlos Aguilar Melchor, Nicolas Aragon, Slim Bettaieb, Loıc Bidoux, Olivier Blazy, Jean-Christophe Deneuville, Philippe Gaborit, Edoardo Persichetti, Gilles Zémor, and IC Bourges. Hamming quasi-cyclic (hqc). *NIST PQC Round*, 2(4):13, 2018.

[mbe] Mbed tls (formerly polarssl). https://tls.mbed.org/. Accessed: 2024-06-16.

[mup] mupq. pqm4 library. https://github.com/mupq/pqm4. Accessed: 2024-06-16.

[Newa] NewAE. Cw1173 chipwhisperer-lite. https://www.newae.com/products/nae-cw1173. Accessed: 2024-06-16.

[Newb] NewAE. Cw520 chipshouter. https://www.newae.com/products/nae-cw520. Accessed: 2024-06-16.

[NIS15] NIST. Sha-3 standard: Permutation-based hash and extendable-output functions. Technical report, U.S. Department of Commerce, Washington, D.C., 2015.

[NIS23a] NIST. Digital signature standard (dss). Technical report, U.S. Department of Commerce, Washington, D.C., 2023.

[NIS23b] NIST. Module-lattice-based digital signature standard. Technical report, U.S. Department of Commerce, Washington, D.C., 2023.

[NIS23c] NIST. Module-lattice-based key-encapsulation mechanism standard. Technical report, U.S. Department of Commerce, Washington, D.C., 2023.

[ope] Openssl. https://www.openssl.org/. Accessed: 2024-06-16.

[OSD04] Charles W Odonnell, G Edward Suh, and Srinivas Devadas. Puf-based random number generation, 2004.

[pca] pq crystals. Dilithium. https://github.com/pq-crystals/dilithium. Accessed: 2024-06-16.

[pcb] pq crystals. Kyber. https://github.com/pq-crystals/kyber. Accessed: 2024-06-16.

[PP21] Lukas Prokop and Peter Peßl. Fault attacks on cca-secure lattice kems. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2021(2):37–60, 2021.

[PQC] PQClean. Pqclean library. https://github.com/PQClean/PQClean. Accessed: 2024-06-16.

[pT] pico Technology. Picoscope 3000 series. https://www.picotech.com/oscilloscope/3000/usb3-oscilloscope-logic-analyzer. Accessed: 2024-06-16.

[QZZG23] Shipei Qu, Xiaolin Zhang, Chi Zhang, and Dawu Gu. Abusing processor exception for general binary instrumentation on bare-metal embedded devices. *arXiv preprint arXiv:2311.16532*, 2023.

[RJH+19] Prasanna Ravi, Mahabir Prasad Jhanwar, James Howe, Anupam Chattopadhyay, and Shivam Bhasin. Exploiting determinism in lattice-based signatures: practical fault attacks on pqm4 implementations of nist candidates. In *Proceedings of the 2019 ACM Asia Conference on Computer and Communications Security*, pages 427–440, 2019.

[RRB+19] Prasanna Ravi, Debapriya Basu Roy, Shivam Bhasin, Anupam Chattopadhyay, and Debdeep Mukhopadhyay. Number not used once-practical fault attack on pqm4 implementations of nist candidates. In *Constructive Side-Channel Analysis and Secure Design: 10th International Workshop, COSADE 2019, Darmstadt, Germany, April 3–5, 2019, Proceedings 10*, pages 232–250. Springer, 2019.

[RRCB20] Prasanna Ravi, Sujoy Sinha Roy, Anupam Chattopadhyay, and Shivam Bhasin. Generic side-channel attacks on cca-secure lattice-based pke and kems. *IACR transactions on cryptographic hardware and embedded systems*, pages 307–335, 2020.

[RYB+23] Prasanna Ravi, Bolin Yang, Shivam Bhasin, Fan Zhang, and Anupam Chattopadhyay. Fiddling the twiddle constants-fault injection analysis of the number theoretic transform. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2023.

[st23a] CRYSTALS-Kyber submission team. Discussion about kyber's tweaked fo transform. Forum post on pqc-forum, 2023. available at https://groups.google.com/a/list.nist.gov/g/pqcforum/c/WFRDl8DqYQ4.

[st23b] CRYSTALS-Kyber submission team. Kyber decisions, part 2: Fo transform. Forum post on pqc-forum, 2023. available at https://groups.google.com/a/list.nist.gov/g/pqc-forum/c/C0D3W1KoINY/m/99kIvydoAwAJ.

[UMB+23] Vincent Quentin Ulitzsch, Soundes Marzougui, Alexis Bagia, Mehdi Tibouchi, and Jean-Pierre Seifert. Loop aborts strike back: Defeating fault countermeasures in lattice signatures with ilp. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2023(4):367–392, 2023.

[wI] wolfSSL Inc. wolfssl embedded ssl library. https://www.wolfssl.com. Accessed: 2024-06-16.

[XIU+21] Keita Xagawa, Akira Ito, Rei Ueno, Junko Takahashi, and Naofumi Homma. Fault-injection attacks against nists post-quantum cryptography round 3 kem candidates. 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 II 27*, pages 33–61. Springer, 2021.