Mind the Faulty KECCAK: A Practical Fault Injection Attack Scheme Apply to All Phases of ML-KEM and ML-DSA

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

***Abstract*—ML-KEM and ML-DSA are NIST-standardized lattice-based post-quantum cryptographic algorithms. In both algorithms, KECCAK is the designated hash algorithm extensively used for deriving sensitive information, making it a valuable target for attackers. In the field of fault injection attacks, few works targeted KECCAK, and they have not fully explored its impact on the security of ML-KEM and ML-DSA. Consequently, many attacks remain undiscovered. In this article, we first identify various fault vulnerabilities of KECCAK that determine the (partial) output by manipulating the control flow under a practical loop-abort model. Then, we systematically analyze the impact of a faulty KECCAK output and propose six attacks against ML-KEM and five attacks against ML-DSA, including key recovery, signature forgery, and verification bypass. These attacks cover the key generation, encapsulation, decapsulation, signing, and verification phases, making our scheme the first to apply to all phases of ML-KEM and ML-DSA. The proposed attacks are validated on the C implementations of the PQClean library's ML-KEM and ML-DSA running on embedded devices. Experiments show that the required loop-abort faults can be realized on ARM Cortex-M0+, M3, M4, and M33 microprocessors with low-cost electromagnetic fault injection settings, achieving a success rate of 89.5%. Once the fault injection is successful, all proposed attacks can succeed with a probability of 100%.**

***Index Terms*—Post-Quantum Cryptography, Fault Injection Attack, KECCAK, ML-KEM, ML-DSA, ARM Cortex-M.**

I. INTRODUCTION

THE National Institute of Standards and Technology (NIST) Post-Quantum Cryptography (PQC) standardization has selected two winners: CRYSTALS-Kyber and CRYSTALS-Dilithium [1], [2]. In 2024, they were further modified and standardized as ML-KEM and ML-DSA [3], [4]. ML-KEM is a lattice-based Key-Encapsulation Mechanism (KEM) that can be used to establish a shared key over a public channel between two parties. ML-DSA is a lattice-based digital signature that can detect unauthorized changes to the data and authenticate the signatory's identity.

The security of ML-KEM and ML-DSA is based on the presumed hardness of the Module Learning-with-Errors (MLWE) problem [5] (ML-DSA is also based on the MSIS problem). The task of the MLWE problem is to solve a set of "noisy" linear equations over a polynomial ring. Informally, it can be described as: Assume $\mathbf{A}$ is an $n \times m$ matrix over a polynomial ring, and $\mathbf{s}$ and $\mathbf{e}$ are vectors of lengths m and n, respectively. Given the equation system $\mathbf{t} = \mathbf{A} \cdot \mathbf{s} + \mathbf{e}$, where $\mathbf{A}$ and $\mathbf{t}$ are known, $\mathbf{e}$ is unknown, but its coefficients are known to be small. The task of MLWE is to solve for $\mathbf{s}$, which is believed to be difficult even against adversaries with a quantum computer.

As standard cryptographic algorithms in the era of quantum computing, ML-KEM and ML-DSA will be widely deployed in scenarios like the Internet of Things (IoT), industrial automation, and automotive electronics. However, although these algorithms are currently believed to be secure, their specific implementations may be compromised by physical attacks, such as Fault Injection Attacks (FIA). FIA induces faults during the execution of a cryptographic algorithm and causes unexpected outputs that reveal sensitive information. Especially for embedded devices often used in the above scenarios, their frequent deployment in (semi-)public locations makes them more physically accessible to FIA attackers. Therefore, it is important to analyze the vulnerability of ML-KEM and ML-DSA implementations to FIAs.

A. Related Works and Motivation

The existing FIAs on ML-KEM and ML-DSA (as well as Kyber and Dilithium) can be roughly divided into four types.

The first type generates a weak MLWE instance that exposes the secret key. Espitau et al. zeroize part of the noise $\mathbf{e}$ to recover $\mathbf{s}$ [6]. Ravi et al. inject multiple faults during the sampling of $\mathbf{e}$ to make it equal to $\mathbf{s}$, resulting in a solvable equation system $\mathbf{t} = \mathbf{A} \cdot \mathbf{s} + \mathbf{s}$ [7].

The second type of attacks target specific critical operations. For $\mathbf{z} = \mathbf{y} + c \cdot \mathbf{s_1}$ in ML-DSA, where $\mathbf{z}$ and c are part of the signature, $\mathbf{s_1}$ is the private key, and $\mathbf{y}$ is the secret commitment vector. Ravi et al. skip the addition with $\mathbf{y}$ [8], and Espitau et al. set part of $\mathbf{y}$ to zero [6], exposing the private key $\mathbf{s_1}$. Ulitzsch et al. optimized this method to make it applicable to implementations with shuffle countermeasures [9]. Bruinderink et al. proposed a Differential Fault Analysis (DFA) scheme that uses the same $\mathbf{y}$ to sign twice, with correct and faulty c, respectively, to compare the two instances and solve for $\mathbf{s_1}$ [10]. In addition, Xagawa et al. skip the comparative checks to expose the secret key or bypass verification [11].

The third type includes Ineffective Fault Analysis (IFA) and Fault Correction Attacks (FCA). Pessl et al. inject faults into the decoding process of the decapsulation in Kyber and infer information about the key based on whether the decryption was successful (whether the fault is effective) [12]. Hermelink et al. use FCA instead of IFA, inputting invalid ciphertext, injecting bit flip faults, and observing whether the decryption was successful (fault correction) [13]. Delvaux et al. improved this attack and significantly increased the time window for

Yuxuan Wang, Jintong Yu, Shipei Qu, Xiaolin Zhang, Xiaowei Li, Chi Zhang, and Dawu Gu (Corresponding authors: Chi Zhang; Dawu Gu.) are with the School of Electronic Information and Electrical Engineering, Shanghai Jiao Tong University, Shanghai 200240, China and the State Key Laboratory of Cryptology, P. O. Box 5159, Beijing 100878, China (e-mail: 18588297218@sjtu.edu.cn; jintongyu@sjtu.edu.cn; shipeiqu@sjtu.edu.cn; xiaolinzhang@sjtu.edu.cn; happy_lxw@sjtu.edu.cn; zc-sjtu@sjtu.edu.cn; dwgu@sjtu.edu.cn).

fault injection [14]. Similar attacks are also proposed against the signing phase of Dilithium [15]. Because each faulty execution can only leak a small amount of information, such attacks often require thousands of injections to recover the key.

The fourth type of attacks target general components or primitives. Ravi et al. proposed a novel attack targeting the Number Theoretic Transform (NTT) that accelerates multiplication on polynomial rings [16]. This attack achieves key recovery, signature forgery, and verification bypass.

Compared to other types, the fourth type poses a broader threat because it targets more generic components in ML-KEM and ML-DSA, resulting in multiple attack points within the algorithms. Table I summarizes the existing attacks. For the key generation phase of ML-KEM and ML-DSA, the encapsulation and decapsulation phases of ML-KEM, and the signing and verification phases of ML-DSA, most other attacks apply to only one or two phases, as their attack targets only appear in these phases. In contrast, the fourth type applies to most of them and has a larger attack surface that poses a severe threat to more real-world instances. However, only one such attack (the NTT attack) has been proposed [16].

In addition to NTT, KECCAK is a more widely used generic component in all phases of ML-KEM and ML-DSA. KECCAK is the hash function and extendable-output function specified in ML-KEM and ML-DSA. It is extensively used for expanding secret random numbers, sampling secret data, and hashing secret information. Many KECCAK outputs derive sensitive information, making it a valuable attack target. Despite the attention given to KECCAK in Side-Channel Analyses (SCA) related to ML-KEM and ML-DSA [17], [18], only a few works have focused on it in the context of FIAs. Bruinderink et al. injected faults into KECCAK, using it merely to disrupt $\mathbf{z} = \mathbf{y} + c \cdot \mathbf{s_1}$ to perform DFA, making it applicable only to the signing phase [10]. Therefore, we categorize it as the second type. Existing attacks have not explored KECCAK's potential as a generic component, remaining many attacks undiscovered.

Therefore, this article systematically analyze the impact of a faulty KECCAK output on ML-KEM and ML-DSA and propose multiple attacks that form a comprehensive scheme applicable to all phases. This enables attackers to compromise the security of either ML-KEM or ML-DSA regardless of which phase is being executed on the target device. In addition to a broader attack surface, this article also focuses on the attacks' practicality. First, our attacks avoid excessive required faults (such as thousands for the third type) to reduce costs. Second, the KECCAK attacks provide multiple optional injection points to improve the success rate. Third, the required faults are validated for feasibility on real devices.

TABLE I
FIAS ON ML-KEM AND ML-DSA AND THE PHASES THEY APPLY TO

Attacks	Type	KeyGen	Encaps	Decaps	Sign	Verify
[6]	1, 2	✓	×	×	✓	×
[7]	1	✓	✓	×	✓	×
[8]	2	×	×	×	✓	×
[9]	2	×	×	×	✓	×
[10]	2	×	×	×	✓	×
[11]	2	×	×	✓	×	✓
[12]	3	×	×	✓	×	×
[13]	3	×	×	✓	×	×
[14]	3	×	×	✓	×	×
[15]	3	×	×	×	✓	×
[16]	4	✓	✓	×	✓	✓
This Work	4	✓	✓	✓	✓	✓

B. Contributions

1) Customized KECCAK *Attacks:* We propose several KECCAK attacks by manipulating the control flow using loop-abort faults to set its (partial) outputs to pre-known values. Existing attacks against KECCAK itself often recover intermediate states through known outputs and require multiple executions with the same input. However, in ML-KEM and ML-DSA, the KECCAK output is usually unknown, and the input varies between each execution, necessitating our customized attacks aimed at output recovery. Furthermore, our attacks target the control flow rather than the permutations, making them unaffected by countermeasures such as shuffling the permutations.

2) New FIA Scheme on ML-KEM and ML-DSA: This article proposes a new fault attack scheme targeting ML-KEM and ML-DSA, the first to apply to all their phases. We utilize (partially) known KECCAK outputs to solve for sensitive information derived from or computed with it, enabling key recovery, signature forgery, and verification bypass attacks. We present six attacks against ML-KEM and five against ML-DSA, forming a comprehensive scheme. These attacks require injecting faults during only one single execution.

These attacks are quite different from the exploitation of KECCAK SCA [17]. First, our attacks only utilize the recovered output, whereas existing SCA exploits the recovered input. Second, our scheme includes multiple new attacks targeting unanalyzed KECCAK instances. Some interesting attacks only utilize partially recovered faulty outputs and their unique relationship with the inputs. Third, our scheme enables verification bypass attacks that SCA cannot achieve.

3) Practical Attacks on Real-World Devices: Firstly, we demonstrated the practicality of loop-abort faults on five ARM Cortex-M devices from four different series, achieving a success rate of 89.5%. We measured the fault characteristics of these devices under Electromagnetic Fault Injection (EMFI) in terms of spatial location, time, and pulse intensity. We also provide guidance on quickly determining fault injection parameters and triggering faults. Secondly, we validated the proposed attacks with the C implementation of ML-KEM and ML-DSA from the PQClean library[1]. Once the fault injection is successful, we can recover the key, forge signatures, or bypass verification with a 100% probability. Our experimental code is open source[2].

C. Structure of the Paper

Section II introduces the background. Section III provides an overview of our attack scheme. Section IV presents the customized KECCAK attacks. Section V is the main focus, detailing the attacks on ML-KEM and ML-DSA. Section VI

[1]https://github.com/PQClean/PQClean
[2]https://github.com/wyxsjtu/mind-the-faulty-keccak

covers the experimental results. Section VII discusses the countermeasures. Section VIII concludes the article.

II. Background

A. Notation

In this article, $\mathbb{Z}_q$ denotes the ring of integers modulo q. R_q denotes the polynomial ring $\mathbb{Z}_q[X]/(X^n+1)$. T_q denotes the image of R_q under the NTT transform, the set of n tuples over $\mathbb{Z}_q$. R_q^k and $R_q^{k\times l}$ denote the sets of length-k vectors and shape-$k\times l$ matrices of polynomials in R_q. Polynomials, vectors, and 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. Variables with a "hat" (e.g., $\hat{\mathbf{a}}$ and $\hat{\mathbf{A}}$), whose elements are in T_q, denote the NTT form of the corresponding polynomial vector or matrix. $\mathbf{a}^T$ and $\mathbf{A}^T$ denote the transpose of vectors or matrices. Symbol $\leftarrow$ denotes assigning the value of the right side to the left side variable, $||$ denotes concatenation of two bit or byte strings, $\circ$ denotes multiplication in ring T_q, $\cdot$ denotes other multiplications in $\mathbb{Z}$, $\mathbb{Z}_q$ or R_q, and $\perp$ means lack of output.

B. Keccak and SHA-3

Keccak is a family of sponge-based hash functions standardized by NIST as SHA-3. SHA-3 includes the hash functions SHA3-224, SHA3-256, SHA3-384, and SHA3-512, and the XOFs SHAKE128 and SHAKE256. SHA-3 operates on a 1600-bit state, which consists of two parts of sizes r (rate) and c (capacity). The rate r is the size of input and output blocks, and the capacity c determines the maximum security level. SHA-3 consists of absorbing and squeezing. We introduce their process based on the PQClean implementation below.

1) Absorbing Phase: As shown in Algorithm 1, the state s is an array initialized to zero. Each input message block of size r is XORed into the first part of the state. After each block, the state is permuted using a 24-round Keccak-f permutation. This article focuses on the control flow rather than the permutation, so Keccak-f will not be elaborated upon. For the remaining message with a length less than r, copy it byte by byte into a helper array t (lines 7-9), and then pad t according to the Keccak padding rules. Then, t is XORed into the state, eight bytes at a time. The absorbing phase finally returns the updated state.

2) Squeezing Phase: As shown in Algorithm 2, data of length r is read from the state each time until the output length is reached. The loop in lines 2-7 processes the output blocks. Each time, the state is first permuted using Keccak-f, and then the first part of it is appended to the helper array t' in line 5. Therefore, the length of t' is a multiple of r. For the final truncation process, a loop (lines 8-10) copies the required length of t' to the final Keccak output o.

3) The Incremental API: ML-KEM and ML-DSA also use the incremental APIs of SHAKE128 and SHAKE256 defined in SP 800-185 [19]. The Incremental API breaks the complete absorb or squeeze process into multiple steps. In the implementation, the state s includes an additional element to record the number of bytes that have already been absorbed or squeezed but not yet permuted. When this counter reaches the rate, Keccak-f is performed for permutation.

Algorithm 1 Keccak absorbing

Input: Message m, rate r (in bytes)
Output: State s

1: $s \leftarrow [0, 0, \ldots, 0]$ ▷ State (int64) initialized to zero
2: **for** complete message blocks (r bytes of m) **do**
3: State s XOR message block bitwise
4: Permute state s using the 24-round Keccak-f
5: **end for**
6: $t \leftarrow [0, 0, \ldots, 0]$ ▷ Helper byte array t
7: **for** $i = 0$ to (byte length of remaining message) $- 1$ **do**
8: Copy i-th byte of remaining message to $t[i]$
9: **end for**
10: Add the padding bits to t
11: **for** $i = 0$ to $r/8$ **do**
12: State s XOR t bitwise, 8 bytes at a time
13: **end for**
14: **return** s

Algorithm 2 Keccak squeezing

Input: State s, output length $olen$ (in bytes), rate r (in bytes)
Output: Hash result o (a byte array) of length $olen$

1: Define helper byte array t'
2: **for** output blocks (r bytes at a time) **do**
3: Permute state s using the 24-round Keccak-f
4: **for** $i = 0$ to $r/8$ **do**
5: Append 8 bytes of s to byte array t'
6: **end for**
7: **end for**
8: **for** $i = 0$ to $olen - 1$ **do**
9: $o[i] \leftarrow t'[i]$ ▷ The length of t' is a multiple of r.
10: **end for**
11: **return** o

C. ML-KEM

ML-KEM is a NIST standard KEM algorithm derived from Kyber. The Public-Key Encryption (PKE) scheme called K-PKE is constructed from the MLWE problem. Then, K-PKE is converted into the ML-KEM using the Fujisaki-Okamoto (FO) transform [20], which is believed to satisfy the IND-CCA2 security. Table II shows ML-KEM's parameter sets, where n and q are parameters of R_q. ML-KEM consists of three phases: key generation, encapsulation, and decapsulation. Given their complexity, we use a slightly simplified pseudocode, focusing on the key operations and those involved in our attacks.

1) Functions: `Expand` expands seed into a $k\times k$ matrix. $\texttt{SampleCBD}_\eta$ samples $a \in R_q$ with coefficients in $[-\eta, \eta]$. `NTT` and $\texttt{NTT}^{-1}$ convert matrices/vectors to and from the NTT domain. `ekEncode`/`dkEncode` and `ekDecode`/`dkDecode`

TABLE II
Parameter Sets for ML-KEM

Parameter set	n	q	k	η_1	η_2	d_u	d_v
ML-DSA-512	256	3329	2	3	2	10	4
ML-DSA-768	256	3329	3	2	2	10	4
ML-DSA-1024	256	3329	4	2	2	11	5

encode the keys of K-PKE into a byte string and vice versa. `K-PKE.Encrypt/Decrypt` perform the K-PKE encryption and decryption. `mEncode` encodes a bit string into a polynomial. $\texttt{uEncode}_{d_u}$ and $\texttt{vEncode}_{d_v}$ compress and encode the ciphertext components into byte strings using d_u and d_v.

2) Key Generation: Generate the encapsulation key ek and decapsulation key dk. The internal function shown in Algorithm 3 takes two randomness, d and z, generated and checked by the outer function as inputs. Firstly, hash d and k to obtain seeds ρ and σ using SHA3-512. ρ is used to generate the matrix $\hat{\mathbf{A}}$ (line 3), and σ is used to generate the secret vectors $s, e \in R_q^k$ (lines 4-11). For each polynomial, SHAKE256 is first used to extend the seed σ and the integer N, and then use `SampleCBD` to sample the coefficients in $[-\eta_1, \eta_1]$. Next, compute the MLWE instance $\hat{\mathbf{t}} = \hat{\mathbf{A}} \circ \hat{\mathbf{s}} + \hat{\mathbf{e}}$ (line 12). Finally, encode and construct the key pair (ek, dk) (lines 13-15).

Algorithm 3 ML-KEM Internal Key Generation (Simplified)

Input: 32-byte randomness d, 32-byte randomness z
Output: encapsulation key ek, decapsulation key dk

1: $(\rho, \sigma) \leftarrow$ SHA3-512$(d||k)$ ▷ ρ, σ are 32-byte seeds
2: $N \leftarrow 0$
3: $\hat{\mathbf{A}} \leftarrow$Expand$(\rho)$ ▷ Sample ρ into a $k \times k$ matrix
4: **for** $i = 0$ to $k-1$ **do**
5: $\quad \mathbf{s}[i] \leftarrow$SampleCBD$_{\eta_1}$(SHAKE256$(\sigma||N, 128 \cdot \eta_1)$)
6: $\quad N \leftarrow N + 1$
7: **end for**
8: **for** $i = 0$ to $k-1$ **do**
9: $\quad \mathbf{e}[i] \leftarrow$ SampleCBD$_{\eta_1}$(SHAKE256$(\sigma||N, 128 \cdot \eta_1)$)
10: $\quad N \leftarrow N + 1$
11: **end for**
12: $\hat{\mathbf{t}} \leftarrow \hat{\mathbf{A}} \circ$ NTT$(\mathbf{s})+$NTT$(\mathbf{e})$ ▷ MLWE instance
13: $ek_{\text{PKE}} \leftarrow$ ekEncode$(\hat{\mathbf{t}}, \rho)$
14: $dk_{\text{PKE}} \leftarrow$ dkEncode$(\hat{\mathbf{s}})$
15: **return** $(ek = ek_{\text{PKE}}, dk = dk_{\text{PKE}}||ek||$SHA3-256$(ek)||z)$

3) Encapsulation: Generate the shared secret key K and the ciphertext c given the encryption key ek. The internal function shown in Algorithm 4 takes ek and randomness m as inputs. Firstly, m and the hash of ek are used to generate K and r using SHA3-512. K is the shared secret key, and r is input to the K-PKE encryption algorithm with m and ek_{PKE} (equals ek) to generate the ciphertext c. Finally, return K and c. For the K-PKE encryption (Algorithm 5), it first decodes ek_{PKE} and generates $\hat{\mathbf{A}}$. In lines 4-12, vectors $\mathbf{y}, \mathbf{e_1} \in R_q^k$ and polynomial $\mathbf{e_2} \in R_q$ are sampled similarly to the key generation phase, which involves hashing r and N using SHAKE256 and further sampling using `SampleCBD`. Then, compute $\mathbf{u} = \mathbf{A}^T \cdot \mathbf{y} + \mathbf{e_1}$ and $v = \mathbf{t}^T \cdot \mathbf{y} + e_2 + \mu$, where μ is the encoded m. The encoded $\mathbf{u}$ and v form the ciphertext.

Algorithm 4 ML-KEM Internal Encapsulation (Simplified)

Input: Encapsulation key ek, 32-byte randomness m
Output: 32-byte shared secret key K, ciphertext c

1: $(K, r) \leftarrow$ SHA3-512$(m||$SHA3-256$(ek))$
2: $c \leftarrow$ K-PKE.Encrypt(ek, m, r)
3: **return** (K, c)

Algorithm 5 K-PKE.Encrypt (Simplified)

Input: Encryption key ek_{PKE}, 32-byte message m, 32-byte randomness r
Output: Ciphertext c

1: $N \leftarrow 0$
2: $(\hat{\mathbf{t}}, \rho) \leftarrow$ ekDecode(ek_{PKE}) ▷ Decode ek_{PKE}
3: $\hat{\mathbf{A}} \leftarrow$Expand$(\rho)$ ▷ Sample ρ into a $k \times k$ matrix
4: **for** $i = 0$ to $k-1$ **do**
5: $\quad \mathbf{y}[i] \leftarrow$ SampleCBD$_{\eta_1}$(SHAKE256$(r||N, 128 \cdot \eta_1)$)
6: $\quad N \leftarrow N + 1$
7: **end for**
8: **for** $i = 0$ to $k-1$ **do**
9: $\quad \mathbf{e_1}[i] \leftarrow$ SampleCBD$_{\eta_2}$(SHAKE256$(r||N, 128 \cdot \eta_2)$)
10: $\quad N \leftarrow N + 1$
11: **end for**
12: $\mathbf{e_2} \leftarrow$ SampleCBD$_{\eta_2}$(SHAKE256$(r||N, 128 \cdot \eta_2)$)
13: $\mathbf{u} \leftarrow$NTT$^{-1}(\hat{\mathbf{A}}^T \circ$ NTT$(\mathbf{y})) + \mathbf{e_1}$
14: $\mu \leftarrow$mEncode(m) ▷ Encode m into a polynomial
15: $v \leftarrow$NTT$^{-1}(\hat{\mathbf{t}}^T \circ$ NTT$(\mathbf{y})) + e_2 + \mu$
16: **return** $c = (\text{uEncode}_{d_u}(\mathbf{u})||\text{vEncode}_{d_v}(v))$

4) Decapsulation: This phase decapsulates the ciphertext c to obtain the shared secret key K using the decryption key dk. As shown in Algorithm 6, m' is obtained by decrypting c with dk_{PKE}. Then, m' and h are used to generate 32-byte arrays K' and r' using SHA3-512 (line 3). Next, generate a pseudorandom array $\bar{K}$ by hashing randomness z and ciphertext c using SHAKE256 (line 4). The FO procedure also involves a re-encryption step, which encrypts the recovered m' with ek_{PKE} and r', obtaining c'. If c' is not equal to c, perform the "implicit rejection": K' is replaced with the pseudorandom $\bar{K}$. This ensures security when the higher level protocols fail to check the return value. Finally, return the shared secret K'.

Algorithm 6 ML-KEM Internal Decapsulation (Simplified)

Input: Decapsulation key dk, ciphertext c
Output: 32-byte shared secret key K'

1: $(dk_{\text{PKE}}, ek_{\text{PKE}}, h, z) \leftarrow$ dkDecode(dk) ▷ h is hash of ek
2: $m' \leftarrow$K-PKE.Decrypt(dk_{PKE}, c) ▷ Decrypt ciphertext
3: $(K', r') \leftarrow$ SHA3-512$(m'||h)$
4: $\bar{K} \leftarrow$ SHAKE256$(z||c, 32)$ ▷ $\bar{K}$ is pseudorandom
5: $c' \leftarrow$ K-PKE.Encrypt$(ek_{\text{PKE}}, m', r')$ ▷ Re-encryption
6: **if** $c' \neq c$ **then**
7: $\quad K' \leftarrow \bar{K}$ ▷ implicitly reject
8: **end if**
9: **return** K'

D. ML-DSA

ML-DSA is a NIST standard digital signature algorithm derived from Dilithium. The ML-DSA scheme uses the Fiat-Shamir With Aborts construction [21]. The security of ML-DSA is based on the MLWE problem. Table III shows ML-DSA's three parameter sets: ML-DSA-44, ML-DSA-65, and ML-DSA-87, all of them use same $n = 256$ and $q = 8380417$ for R_q. ML-DSA has three phases: key generation, signing, and verification. We also use the simplified pseudocode.

TABLE III
PARAMETER SETS FOR ML-DSA

Parameter set	(k,l)	d	τ	λ	γ_1	γ_2	η
ML-DSA-44	(4,4)	13	39	128	2^{17}	8380416/88	2
ML-DSA-65	(6,5)	13	49	192	2^{19}	8380416/32	4
ML-DSA-87	(8,7)	13	60	256	2^{19}	8380416/32	2

1) Functions: `Expand` function expands a 32-byte seed into a $k \times l$ polynomial matrix. `Sample`, `ExpandMask`, and `SampleInBall` are sampling fuctions based on SHAKE256, generating polynomials or vectors with coefficients in $[-\eta, \eta]$, $[-\gamma_1 + 1, \gamma_1]$, and $\{-1, 0, 1\}$, respectively. They process XOF's output to generate sampling results. `NTT` and $\texttt{NTT}^{-1}$ convert matrices/vectors to and from the NTT domain. `pkEncode`/`skEncode`/`sigEncode` and `pkDecode`/`skDecode`/`sigDecode` encode public/secret keys or signatures into a byte string and vice versa. Rounding functions: `Power2Round` decomposes r into (r_1, r_0) s.t. $r = 2^d \cdot r_1 + r_0 \bmod q$. `HighBits` extracts the high bits of the coefficients and `UseHint` adjusts them using a hint h.

2) Key Generation: This phase generates the public key pk and secret key sk. The ML-DSA's key generation is similar to that of ML-KEM. The internal function (Algorithm 7) takes a randomness ξ as input. Firstly, seeds ρ, ρ', and randomness K are generated using SHAKE256. ρ is used to sample the matrix $\hat{\mathbf{A}}$ (line 2), and ρ' is used to sample the secret polynomial vectors $\mathbf{s_1} \in R_q^l$ and $\mathbf{s_2} \in R_q^k$ (lines 3-8). In the `Sample` function, ρ' and r are absorbed into the SHAKE256 state, and the polynomials are generated using the squeeze output. Next, compute the MLWE instance $\mathbf{t} = A \cdot s_1 + s_2$ in line 9. Finally, decompose $\mathbf{t}$ into higher/lower bits and construct (pk, sk).

Algorithm 7 ML-DSA Internal Key Generation (Simplified)

Input: 32-byte randomness ξ
Output: public key pk, secret key sk
1: $(\rho, \rho', K) \leftarrow$ SHAKE256($\xi||k||l$,128) ▷ ρ, ρ', K are 32, 64, 32 bytes long, respectively
2: $\hat{\mathbf{A}} \leftarrow$Expand($\rho$) ▷ Sample ρ into a $k \times l$ matrix
3: **for** $r = 0$ to $l - 1$ **do**
4: $\quad \mathbf{s_1}[r] \leftarrow$ Sample($\rho'||r$) ▷ Based on SHAKE256
5: **end for**
6: **for** $r = 0$ to $k - 1$ **do**
7: $\quad \mathbf{s_2}[r] \leftarrow$ Sample($\rho'||r + l$) ▷ Based on SHAKE256
8: **end for**
9: $\mathbf{t} \leftarrow \text{NTT}^{-1}(\hat{\mathbf{A}} \circ \text{NTT}(\mathbf{s_1})) + \mathbf{s_2}$ ▷ MLWE instance
10: $(\mathbf{t_1}, \mathbf{t_0}) \leftarrow$Power2Round($\mathbf{t}$) ▷ Split high/low-order bits
11: $pk \leftarrow$pkEncode($\rho, \mathbf{t_1}$)
12: $tr \leftarrow$SHAKE256(pk, 64)
13: $sk \leftarrow$ skEncode($\rho, K, tr, \mathbf{s_1}, \mathbf{s_2}, \mathbf{t_0}$)
14: **return** (pk, sk)

3) Signing: Sign a signature σ for a message using the secret key sk. The internal signing (Algorithm 8) takes sk, formatted message M', and rnd as inputs. the hedged variant of ML-DSA (default) uses a random rnd, while the deterministic variant uses a fixed value. First, decode sk and compute $\hat{\mathbf{A}}$ and the message representative μ (lines 1-4). The private random seed ρ'' is the SHAKE256 hash value of K, rnd, μ (line 5). The following is the abort loop of the Fiat-Shamir construction (lines 7-16). For each iteration, a random $\mathbf{y}$ is generated using the SHAKE256-based `ExpandMask` (line 8). Then, $\tilde{c}$ is generated by hashing μ and encoded $\mathbf{w_1}$ (computed in line 9) using SHAKE256 (line 10). c is sampled by the SHAKE256-based `SampleInBall` function using $\tilde{c}$ (line 11). Next, compute $\mathbf{z} = \mathbf{y} + c \cdot \mathbf{s_1}$. Then, do the validity checks and generate a hint $\mathbf{h}$. If checks fail, update the counter κ and re-execute the abort loop. Finally, σ consists of $\tilde{c}$, $\mathbf{z}$, and $\mathbf{h}$.

Algorithm 8 ML-DSA Internal Signing (Simplified)

Input: Secret key sk, formatted message M', per message randomness or dummy variabel rnd
Output: Signature σ
1: $(\rho, K, tr, \mathbf{s_1}, \mathbf{s_2}, \mathbf{t_0}) \leftarrow$ skDecode(sk)
2: $\hat{\mathbf{s}}_1, \hat{\mathbf{s}}_2, \hat{\mathbf{t}}_0 \leftarrow$ NTT($\mathbf{s_1}$), NTT($\mathbf{s_2}$), NTT($\mathbf{t_0}$),
3: $\hat{\mathbf{A}} \leftarrow$Expand($\rho$) ▷ Sample ρ into a $k \times l$ matrix
4: $\mu \leftarrow$SHAKE256($tr||M'$, 64) ▷ Message representative
5: $\rho'' \leftarrow$ SHAKE256($K||rnd||\mu$, 64) ▷ Random seed
6: $\kappa \leftarrow 0$ $(\mathbf{z}, \mathbf{h}) \leftarrow \perp$ ▷ Init for the abort loop
7: **while** $(\mathbf{z}, \mathbf{h}) = \perp$ **do**
8: $\quad \mathbf{y} \in R_q^l \leftarrow$ExpandMask($\rho'', \kappa$) ▷ SHAKE256-based
9: $\quad \mathbf{w_1} \leftarrow$HighBits($\text{NTT}^{-1}(\hat{\mathbf{A}} \circ \text{NTT}(\mathbf{y}))$)
10: $\quad \tilde{c} \leftarrow$SHAKE256($\mu||$w1Encode($\mathbf{w_1}$), $\lambda/4$)
11: $\quad c \leftarrow$SampleInBall($\tilde{c}$) ▷ Involving SHAKE256
12: $\quad \hat{c} \leftarrow$NTT(c)
13: $\quad \mathbf{z} \leftarrow \mathbf{y} + \text{NTT}^{-1}(\hat{c} \circ \hat{\mathbf{s}}_1)$
14: $\quad$ Generate hint $\mathbf{h}$ and do validity checks, set $(\mathbf{z}, \mathbf{h}) = \perp$ if checks fail
15: $\quad \kappa \leftarrow \kappa + l$
16: **end while**
17: **return** $\sigma =$ sigEncode($\tilde{c}, \mathbf{z}, \mathbf{h}$)

4) Verification: As shown in Algorithm 9, this phase checks whether a signature σ is valid for a message M'. Firstly, decode pk and signature σ, then recover $\hat{\mathbf{A}}$, tr and μ (lines 1-6). Then, use $\tilde{c}$ to sample c through the `SampleInBall` function based on SHAKE256 (line 7). After that, compute w_1' using the signature and public key (lines 8-9). In line 10, hash μ and the encoded $\mathbf{w_1'}$ using SHAKE256 to obtain the recovered $\tilde{c}'$. Finally, return whether $\tilde{c}' = \tilde{c}$ and $\mathbf{z}$ is valid.

Algorithm 9 ML-DSA Internal Verification (Simplified)

Input: Public key pk, formatted message M', signature σ
Output: Boolean verification result
1: $(\rho, \mathbf{t_1}) \leftarrow$ pkDecode(pk)
2: $(\tilde{c}, \mathbf{z}, \mathbf{h}) \leftarrow$ sigDecode(σ)
3: Return False if $\mathbf{h}$ is $\perp$
4: $\hat{\mathbf{A}} \leftarrow$Expand($\rho$) ▷ Sample ρ into a $k \times l$ matrix
5: $tr \leftarrow$SHAKE256(pk, 64)
6: $\mu \leftarrow$SHAKE256($tr||M'$, 64)
7: $c \leftarrow$SampleInBall($\tilde{c}$) ▷ Involving SHAKE256
8: $\mathbf{w}'_{Approx} \leftarrow \text{NTT}^{-1}(\hat{\mathbf{A}} \circ \text{NTT}(\mathbf{z}) - \text{NTT}(c) \circ \text{NTT}(\mathbf{t_1} \cdot 2^d))$
9: $\mathbf{w}_1' \leftarrow$UseHint($\mathbf{h}, \mathbf{w}'_{Approx}$) ▷ Recover $\mathbf{w}_1$
10: $\tilde{c}' \leftarrow$SHAKE256($\mu||$w1Encode($\mathbf{w_1'}$), $\lambda/4$)
11: **return** Whether $\tilde{c}' = \tilde{c}$ **and** $\mathbf{z}$ is valid

III. Attack Scheme Overview

Overall, the proposed attack scheme manipulates the control flow of KECCAK in ML-KEM or ML-DSA through loop-abort faults, making its output known to the attacker. This allows the attacker to subsequently perform key recovery attacks, signature forgery attacks, and verification bypass attacks through further analysis, covering all phases of ML-KEM and ML-DSA. The attack scheme consists of the following three layers:

1) Layer 0 (the attacker model): Inducing loop-abort faults on a real device. Assume that the attacker has physical access to the device. The algorithm running on the device can be **any** phase of either ML-KEM or ML-DSA. Assume the attacker can invoke the algorithm (the outer function) and inject a loop-abort fault during execution. Loop-abort means entirely skipping or prematurely ending a loop. Our experiments in VI demonstrate that this type of fault can be induced with a significant probability through EMFI.

2) Layer 1: Manipulating the control flow of KECCAK to recover its output. The KECCAK implementations involve a number of loops that copy data between arrays or update the state. Applying loop-abort faults to them enables zeroizing crucial arrays or leaving the state un-updated. For the zeroized arrays, the attacker can compute the corresponding KECCAK output derived from them, which are fixed values known in advance. For the state not updated by KECCAK-f, the one-way property is undermined, resulting in a partially recovered faulty output with a unique relationship with the inputs.

3) Layer 2: Attacking ML-KEM and ML-DSA. The KECCAK output is extensively used to derive or sample sensitive information in ML-KEM and ML-DSA, such as the secret random seeds, the polynomials of secret keys, and the shared secret key. Therefore, the Layer 1 KECCAK attacks enable multiple interesting new attacks against ML-KEM and ML-DSA and provide new approaches for some existing attacks.

IV. Layer 1: Fault Vulnerabilities of Keccak

This section presents the vulnerabilities of KECCAK under the loop-abort model based on the PQClean software implementation and proposes the attacks shown in Figure 1.

1) Zeroizing Crucial Arrays: We discover that for the the absorbing phase (Algorithm 1) of KECCAK instances with short inputs, skipping critical assignments would zeroize the state or other crucial arrays, resulting in deterministic outputs. We suppose the input length is less than the rate r, a common situation in ML-KEM and ML-DSA. For example, the input of the KECCAK instance that expand the random seed is typically 32 bytes. This means the loop in lines 2-5 will not be executed. One may also abort this loop to satisfy this situation. We propose the following two attacks:

Attack 1: Abort the loop in lines 7-9 in Algorithm 1, setting array t to zero. The state s is then determined since only the padded t is XORed to it. Therefore, the KECCAK output is set to a fixed value known in advance.

Attack 2: Abort the loop in lines 11-13 in Algorithm 1, making t fail to be XORed to s. Therefore, the state s remains the initial value (zero), resulting in a pre-known output.

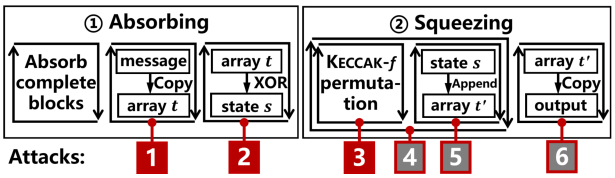


Fig. 1. Attack points of proposed attacks on KECCAK.

2) Skipping the Permutations: We discover that skipping permutations during the squeezing phase (Algorithm 2) results in a partially recovered faulty output related to the inputs. We suppose the input and output length are less than the rate r, meaning that only one squeezing loop (lines 2-7) and one KECCAK-f permutation is involved. All fixed-length output KECCAK members and some SHAKE256 instances in ML-KEM and ML-DSA have outputs shorter than r.

Attack 3: Skip the 24-round KECCAK-f permutation in line 3 of Algorithm 2, making the final state equal to the padded input, with the remaining bits set to zero. This results in interesting characteristics. First, part of the KECCAK output can be recovered: the zero bits, padding bits, and bits corresponding to the known input parts. Second, the unrecovered bits are constrained to be equal to the input.

3) Other "Unstable" Attacks: These attacks skip the update of an uninitialized array, keeping it at a random value. Therefore, these attacks are less recommended and considered as alternatives. However, in some implementations (e.g., those using data structures with initialization) or under specific compilation settings, these arrays may be set to zero, resulting in a fixed, pre-known output. In the squeezing phase (Algorithm 2), these attacks include aborting the outer squeezing loop in lines 2-7 (**Attack 4**), aborting the loop appending output blocks to helper array t' in lines 7-9 (**Attack 5**), and aborting the loop copying data to the output buffer in lines 11-13 (**Attack 6**).

4) Attacking the Incremental API: For the Incremental API, a single attack can only affect one call. The attacker needs to target all absorbing or squeezing calls. However, in practical scenarios, there are more clever handling methods. For example, if there are two absorbing calls, and one of them has a known input, the attacker can attack only the unknown one. Alternatively, if there is only one absorbing call but multiple squeeze calls (common in sampling), apply the attacks on the absorbing phase.

5) Difference from Existing Attacks: The existing fault attacks on KECCAK are mainly Differential Fault Attacks (DFA) and Algebraic Fault Attacks (AFA), which aim to recover the state by modifying the data through bitflips or byte faults [22]–[25]. These attacks need to know the output and require dozens of faulty executions with the same input. However, in ML-KEM and ML-DSA, the randomness leads to different KECCAK inputs in each execution, and the complete output is usually not exposed to the attacker. Therefore, existing attacks are not suitable for these scenarios. In contrast, the proposed attacks are customized for ML-KEM and ML-DSA, which aim to recover the KECCAK by manipulating the control flow through loop-abort faults. Our attacks only need a single execution and do not require knowledge of the output.

V. Layer 2: Attacks on ML-KEM and ML-DSA

This section is the core of our attack scheme. Based on the Layer 1 attacks, we systematically analyze the C implementation of ML-KEM and ML-DSA in the PQClean library and propose several interesting new attacks covering all phases.

A. Key Recoveries and Signature Forgeries on Key Generation

The key generation of ML-KEM and ML-DSA are similar, so they share similar fault vulnerabilities.

1) ML-KEM Attack 1 & ML-DSA Attack 1: Perform the KECCAK attacks on the SHA3-512 or SHAKE256 instance that expands the initial randomness into several random seeds, including the secret seed: σ in ML-KEM and ρ' in ML-DSA (see line 1 in Algorithm 3 and Algorithm 7). This instance has not been analyzed in terms of both FIA and SCA. We provide a new method for recovering secret seeds. The secret seed is used to sample the secret vector in the MLWE instance and is further derived into the private key. Therefore, recovering this KECCAK output leads to key recovery, further enabling shared secret key recovery and signature forgery attacks. The target KECCAK instances' input and output lengths are both less than the rate, satisfying assumptions for all Layer 1 attacks.

For KECCAK Attacks 1 and 2, the KECCAK output and the secret key derived from it are set to fixed, pre-known values.

For KECCAK Attack 3 that recovers partial output, the faulty output equals part of the input after padding. Therefore, the first 32 bytes are the initial randomness d or ξ. For ML-KEM, the 33rd byte is parameter k, the 34th byte is a fixed 0x06 (related to SHA3-512), and the remaining bytes are 0. For ML-DSA, the 33rd and 34th bytes are parameters k and l, the 35th is a fixed 0x1F (related to SHAKE256), and the rest are 0. Therefore, the KECCAK output has only the first 32 bytes unknown, which are precisely the public seed, while the secret seed starting from the 33rd byte is known. For example, for ML-KEM 512, the secret seed is fixed at 0x020600...0.

2) ML-KEM Attack 2 & ML-DSA Attack 2: These attacks target the SHAKE256-based sampling of the secret vectors s and s_1 (see lines 4-7 in Algorithm 3 and lines 3-5 in Algorithm 7). Unlike the SCA targeting these instances to recover the input secret seeds [17], we set the output sampling results to known values. These KECCAK instances use the incremental API. Since only one absorbing call with short input is involved, we target the absorbing phase using KECCAK Attack 1 and 2. This results in the polynomial sampled being set to a fixed and pre-known value. To recover the complete secret vector s or s_1, $k = 2$ faults for ML-KEM-512 and $l = 4$ faults for ML-DSA-44 are needed. For ML-KEM, the NTT form of s is the decapsulation key, which can decrypt the ciphertext and recover the shared secret key. For ML-DSA, s_1 alone is sufficient for signature forgery [10].

3) A New Approach of Existing $\mathbf{e}$ *Attacks:* The work of Espitau et al. [6] zeroizes part of $\mathbf{e}$ of the MLWE instance $\mathbf{t} = \mathbf{A} \cdot \mathbf{s} + \mathbf{e}$, resulting in a weak MLWE instance $\mathbf{t} = \mathbf{A} \cdot \mathbf{s} + [\mathbf{e}_1 \mid 0]$ that exposes the secret vector s. By attacking the KECCAK instances that sample the polynomials of $\mathbf{e}$ (line 9 in Algorithm 3 and line 7 in Algorithm 7), we set the coefficients to known values instead of zero. The secret key can be solved using lattice reduction techniques.

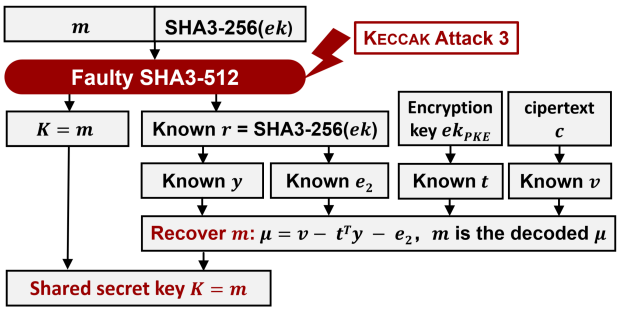


Fig. 2. The process of ML-KEM Attack 3 based on KECCAK Attack 3.

B. Shared Key Recoveries on the Encapsulation of ML-KEM

1) ML-KEM Attack 3: This attack targets the SHA3-512 instance that generates the shared secret key K and randomness r in line 1 of Algorithm 4. For KECCAK Attacks 1 and 2, K and r are set to fixed and pre-known values that can be directly obtained by the attacker. Unlike the SCA that analyzes this instance to recover the sensitive input message m [17], our fault attack directly recovers its output.

More importantly, as shown in Figure 2, we propose a novel attack process that utilizes partially recovered faulty outputs and their unique relationship with the inputs to recover the shared key. The first step is to perform KECCAK Attack 3, making the output of the SHA3-512 instance equal to the padded input. Therefore, the 32-byte shared key K equals the randomness m, the first 32 bytes of the input. The random seed r equals the following 32 bytes of the input, which is the hash of the encapsulation key ek (a known value). The second step is to recover m from the ciphertext using the known seed r. In the K-PKE encryption (Algorithm 5), the random seed r is used to sample y (lines 4-7) and e_2 (line 12), so their values are also known. Then, we can recover μ, the encoded m, by:

$$\mu = v - \mathbf{t}^T \cdot \mathbf{y} - e_2 \tag{1}$$

where v is part of the ciphertext, and $\mathbf{t}$ can be obtained from the public encryption key. Next, decode μ to obtain m. Finally, we recover the faulty shared secret key K that equals m.

2) ML-KEM Attack 4: Attack the SHAKE256-based sampling of vector y in line 5 of Algorithm 5. Similar to ML-KEM Attack 2, we set the polynomials of y to fixed and pre-known values. The y of ML-KEM has 2 polynomials, requiring 2 faults. With the recovered y, we have:

$$\mu' = v - \mathbf{t}^T \cdot \mathbf{y} = \mu + e_2 \tag{2}$$

m can be then recovered by decoding μ'. Though a error of e_2 is introduced, it is smaller than the error of a valid decryption, so the recovered m is correct [26]. Finally, The shared secret key K can be recovered by hashing m and the hash of ek.

3) Attack Scenarios: Assume that Alice and Bob wish to use ML-KEM to establish a shared key K for symmetric encryption of their subsequent communications. Alice first generates the key pair (ek, sk), and sends the encryption key ek to Bob. Bob executes the encapsulation phase. At the same time, an attacker injects faults during the execution and performs ML-KEM Attack 3 or 4, recovering the faulty shared

secret key K^*. Bob then sends the faulty ciphertext to Alice. The decapsulation of Alice fails since a re-encryption step is involved, and the faulty ciphertext cannot pass the check (line 6 of Algorithm 6). However, whether the decapsulation succeeds or not, once Bob encrypts some secret message M with K^* using symmetric encryption (e.g., AES), the attacker can decrypt the ciphertext and obtain M.

Another scenario is the fault-assisted Man-In-The-Middle (MITM) framework proposed by Ravi et al. [16]. This scenario additionally requires the attacker can impersonate as Alice and Bob. The attacker first recovers the seed m and the faulty shared secret key K^* by faulting Bob's encapsulation, and then reconstructs a valid ciphertext and the shared key K using m. Next, the attacker sends the ciphertext to Alice, who decapsulates and obtains K. Finally, the attacker can decrypt all the communications using K for Alice and K^* for Bob.

C. Shared Key Recoveries on the Decapsulation of ML-KEM

We not only propose a method to recover the shared key K' but also introduce an interesting ML-KEM Attack 6, which exploits vulnerabilities of the implicit rejection mechanism.

1) ML-KEM Attack 5: Attack the SHA3-512 instance that generates the shared secret key K' and randomness r' in line 1 of Algorithm 6. We consider KECCAK Attacks 1 and 2, which set K' and r' to fixed and pre-known values. However, the faulty r' leads to decapsulation failure since the re-encryption gets a c' differnt from the original c. Therefore, we make use of the attack proposed by Xagawa et al. [11] that skips the equality check in line 6, making the recovered K' the final shared secret key.

2) ML-KEM Attack 6: ML-KEM employs "implicit rejection" to enhance its security, meaning that a secret random value $\bar{K}$ is returned when decapsulation fails. However, this makes the decapsulation party unable to determine whether the shared key is valid or replaced by $\bar{K}$, which could allow them to encrypt secret messages with $\bar{K}$. This is usually secure since $\bar{K}$ is secret. However, considering fault attacks, we propose a novel attack, which performs KECCAK Attacks 1 or 2 on the SHAKE256 instance used to generate $\bar{K}$ (line 4 of Algorithm 6), setting it to a fixed and pre-known value. To handle the long input of this instance, we can inject faults only during the incremental absorb process of the first part $\mathbf{z}$ and recover the internal state of KECCAK since c can be obtained. Additionally, it needs to make the decapsulation fail and return $\bar{K}$, which can be easily achieved by attacking any KECCAK instance before the equality check in line 6.

3) Attack Scenario: Once the the shared secret key (or the return of the implicit rejection) is used as a symmetric key to encrypt a secret message M, the attacker can obtain the ciphertext from the public channel and recover M.

D. Signature Forgeries on the Signing of ML-DSA

The core of the signing phase is computing $\mathbf{z} = \mathbf{y} + c \cdot \mathbf{s_1}$, where $\mathbf{z}$ and c are parts of the signature, $\mathbf{y}$ is a random vector, and $\mathbf{s_1}$ is the secret vector. Therefore, $\mathbf{y}$ and c are valuable targets for fault attacks. Faulty $\mathbf{y}$ or c may lead to the explosure of $\mathbf{s_1}$, thereby enabling signature forgery.

The existing $\mathbf{y}$ attacks abort the sampling of its polynomials or skip the addition that adds $\mathbf{y}$ to $c \cdot \mathbf{s_1}$ [6], [16], [27]. Their attack points are in the the abort loop of the Fiat-Shamir construction (lines 7-16 of Algorithm 8). This requires their faults to be induced in the last iteration; otherwise, $\mathbf{y}$ is regenerated, rendering the attack ineffective. Therefore, attackers must fault multiple executions (≈ 3 for [16]) or rely on side-channel traces to assist in locating the last iteration. To address this issue, we provide the following attack.

1) ML-DSA Attack 3: This attack targets the secret random seed ρ'' that derives $\mathbf{y}$ instead of $\mathbf{y}$ itself. Perform KECCAK Attacks 1 or 2 on the SHAKE256 instance used to generate ρ'' (line 5 of Algorithm 8), setting it to a fixed and pre-known value. The generation of ρ'' is out of the abort loop (lines 7-16), so this attack only needs faulting one execution. With the known ρ'', the attacker can guess the counter κ, calculate the corresponding $\mathbf{y}$ offline, and then compute $\mathbf{s_1} = c^{-1} \cdot (\mathbf{z} - \mathbf{y})$. We conducted 100 random tests; on average, the attacker can obtain the correct $\mathbf{s_1}$ with 4.16 guesses.

2) Implementation Methods for DFA: Our KECCAK attacks also provide more implementation options for the DFA against the deterministic ML-DSA [10]. One can apply any Layer 1 attack to the SHAKE256 instance in line 10 or the SHAKE256-based sampling in line 11 to produce a faulty c^* and the corresponding $\mathbf{z}^*$. Finally, recover the secret vector by computing $\mathbf{s_1} = (c^* - c)^{-1} \cdot (\mathbf{z}^* - \mathbf{z})$.

E. Verification Bypasses on the Verification of ML-DSA

Because the FIA can actively generate faulty intermediate values, it enables verification bypass attacks that passive SCA cannot achieve. Verification bypass attacks aim to force the acceptance of an invalid signature for any message. Bindel et al. [28] proposed an verification bypass attack against GLP and BLISS by zeroizing the challenge c. The attack of Ravi et al. [16] faults the NTT of c to zero in Dilithium. We propose the following new verification bypass attacks.

1) ML-DSA Attack 4: Instead of zeroizing the c or NTT(c), we zeroize the internal state of KECCAK using KECCAK Attacks 1 or 2, therby producing a fixed and pre-known value for the challenge c. The target is the SHAKE256 instance within the `SampleInBall` function (line 7 of Algorithm 9). With the pre-known c, the attacker can compute the corresponding $\tilde{c}'$ in line 10. Therefore, for a malicious signature $(\tilde{c}, \mathbf{z}, \mathbf{h})$, the attacker sets $\tilde{c}$ to the pre-known $\tilde{c}'$, and $\mathbf{z}, \mathbf{h}$ are any values with valid norms. It can then pass the equality check and norm check in line 11 when the fault is successfully induced.

The key to the attack is to tamper with the value of $\tilde{c}'$, so the attack target is not limited to c. The generation process of $\tilde{c}'$ itself is also an interesting target, yet no existing attacks have targeted it. We propose the following attack.

2) ML-DSA Attack 5: Attack the SHAKE256 instance that generates $\tilde{c}'$ (line 10 of Algorithm 9), setting it to a fixed and pre-known value. This makes a malicious signature with $\tilde{c}$ equal to the pre-known $\tilde{c}'$ bypass the verification. However, a challenge of this attack is that the input length of the target SHAKE256 exceeds the rate, which requires skipping the loop that absorbs complete message blocks through faults in addition to KECCAK attacks 1 or 2.

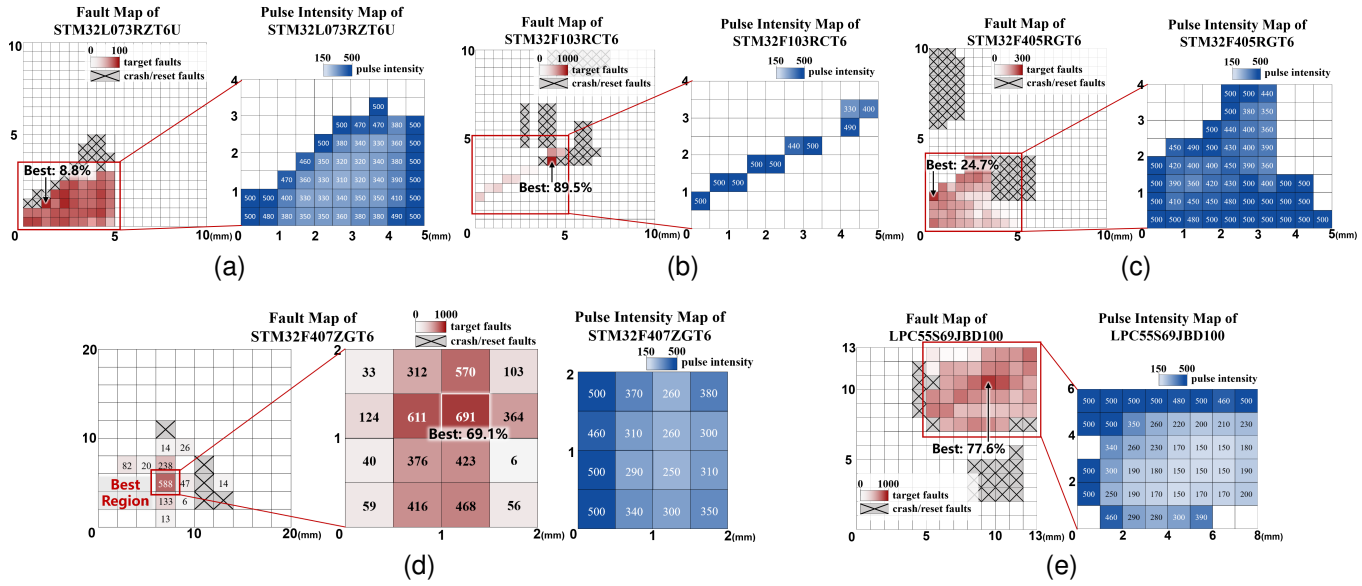


Fig. 3. Space and pulse intensity fault characterization for (a) STM32L073RZT6U, (b) STM32F103RCT6, (c) STM32F405RGT6, (d) STM32F407ZGT6, and (e) LPC55S69JBD100. The left part of each subfigure represents the MCUs' spatial fault characteristics. The red cells indicates that a loop-abort fault can be induced at this location, with darker colors representing higher success rates. For cells with a "×", there are only crashes or resets. The right part is the pulse intensity map for the maximum fault rate. The numbers in the cells (measured in V) represent each position's optimal pulse intensity.

VI. Experimental Validation

ARM Cortex-M is one of the most common embedded processor architectures. This section first demonstrates that loop-abort faults can be induced through EMFI on devices from various ARM Cortex-M series, indicating that our scheme is a powerful attack against embedded ML-KEM and ML-DSA instances. For practicality, we provide detailed fault characterization, fault explanation, and inject point localization guidance. Finally, we present experimental validation of the attack scheme. Our experiment code is open source.

A. Experimental Setup

1) Target Implementation: Our experiments target the C implementation of ML-KEM and ML-DSA in the PQClean library (the "clean" implementation).

2) Devices Under Test (DUTs): We select five ARM Cortex-M MCUs from four different series, covering devices from low performance to high performance, single-core to multi-core: the Cortex-M0+ STM32L073RZT6U, the Cortex-M3 STM32F103RCT6, the Cortex-M4 STM32F405RGT6, the Cortex-M4 STM32F407ZGT6, and the Cortex-M33 dual core NXP LPC55S69JBD100.

3) Experimental Environment: We use EMFI to implement our attacks. The environment consists of a controller PC, the DUT, a NewAE ChipWhisperer, a NewAE ChipSHOUTER, and an oscilloscope. The controller PC communicates with the DUT using a UART connection, instructing it to execute the target code. Before executing the target operation, the DUT activates a trigger signal. The ChipWhisperer precisely controls the time offset from the trigger signal, and the ChipSHOUTER generates the EM pulse for fault injection. The ChipSHOUTER is mounted on an XYZ table for fine-grained spatial adjustments. We also use an oscilloscope to assist in fault locating. The PC collects the final output.

B. Fault Characterization

This section presents the fault characterization of the five DUTs in the spatial (different positions), time (different offsets), and pulse intensity dimensions. We adopt a simple iterative array assignment from the PQClean KECCAK implementation as the target. A trigger signal is activated right before the target loop. We use a single pulse with a width of 100 ns for fault injection, and the EM probe is closely attached to the MCU package.

1) Spatial and Pulse Intensity Fault Characterizations: We divide the MCU package into 0.5mm square grid cells. We first determine a time offset that can trigger a fault with a considerable probability. For each grid cell, we inject faults at this time point and adjust the intensity of the EM pulse to achieve the highest success rate of loop-abort faults. The pulse intensity of the ChipSHOUTER can be adjusted by setting the voltage of the coil, ranging from 150V to 500V. We first conduct 100 injection attempts at each intensity with an interval of 10 V to find the optimal intensity and then perform 1000 experiments at the optimal intensity. Figure 3 shows the results on the five DUTs. The left part of each subfigure displays the fault rate at each position under the optimal pulse intensity. Red cells mean that loop-abort faults are observed, with darker colors representing higher success rates. The optimal positions and their corresponding success rates are marked in the figure, achieving 89.5% on the STM32F103RCT6. Cells with "×" mean that only crash or reset faults were observed at that position, while no color represents no faults. The right part shows the optimal pulse intensity for each cell that can induce loop-abort faults.

We summarize the following characteristics: First, the areas that can induce loop-abort faults are generally concentrated and relatively large, which means attackers do not need precise spatial positioning. Second, the central part of the area does

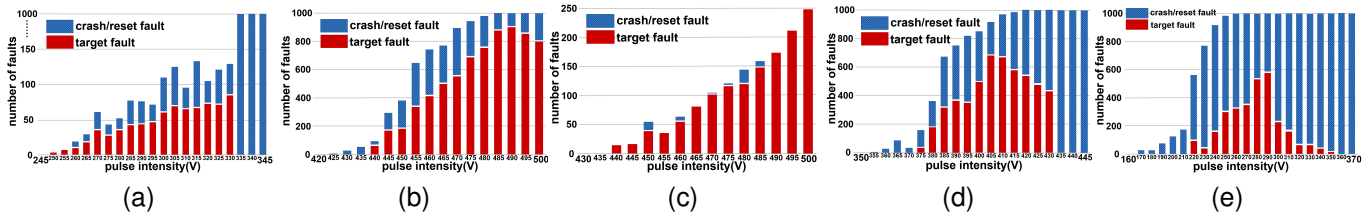


Fig. 4. The number of faults in 1000 injections at different pulse intensities in the optimal position, where the red parts represent loop-abort faults and blue represents crashes or resets. (a) STM32L073RZT6U, (b) STM32F103RCT6, (c) STM32F405RGT6, (d) STM32F407ZGT6, and (e) LPC55S69JBD100.

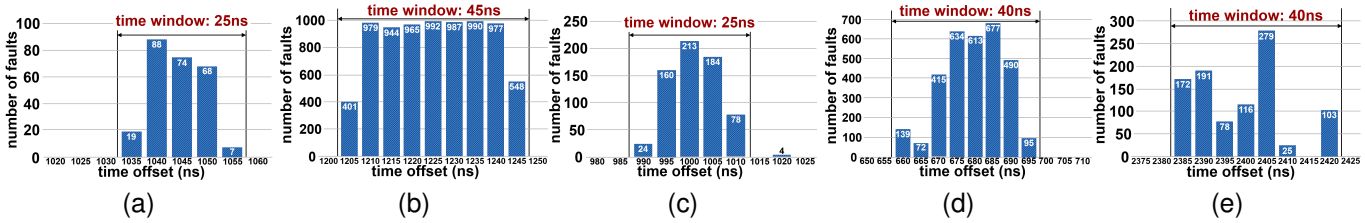


Fig. 5. The number of faults in 1000 injections at different time offsets. (a) STM32L073RZT6U, (b) STM32F103RCT6, (c) STM32F405RGT6, (d) STM32F407ZGT6, and (e) LPC55S69JBD100.

not necessarily have the highest success rate, as it may be easier to crash or reset. Third, the central part of the area requires a smaller pulse intensity than the edges, which may suggest that the affected circuits are located there.

We also explore the impact of pulse intensity on the success rate of loop-abort faults. We conducted 1,000 experiments for each intensity with a 5 V interval at the optimal position, and the results are shown in Figure 4. The red portion of the bars represents loop-abort faults, while the blue represents crashes or resets. We observe that greater intensity can lead to more faults, but it also makes crashes or resets more likely. The success rate of loop-abort faults exhibits an unimodal distribution concerning the pulse intensity. Thus, it is necessary to find a moderate intensity. For the STM32F405 DUT, the pulse intensity corresponding to the peak may exceed 500 V.

2) Time Fault Characterization: We set the core frequency of the DUTs to 16 MHz, select appropriate spatial positions and pulse intensities, and explore the fault rate under different time offsets. We use the Chipwhisperer's 200 MHz clock to achieve a precise time delay with 5 ns accuracy. We perform 1000 fault injection attempts for each offset, and the results are shown in Figure 5. The faults are highly time-sensitive and can only be induced within a 25-45 ns time window. It is in the same order of magnitude as the 62.5 ns duration of a single instruction at a 16 MHz clock frequency.

3) A Guide for Inducing Loop-Abort Faults: To enhance the practicality of our attacks in real-world scenarios, we provide a heuristic guide based on fault characterization for quickly identifying fault parameters and inducing loop-abort faults on ARM Cortex-M devices. Step 1: Scan spatially to determine the boundaries of areas susceptible to EM pulses since these areas are often concentrated and continuous. In this step, we use the maximum pulse intensity and an approximate offset (500-2500 ns). Step 2: Precisely locate the time offset. Use moderate pulse intensity within the area and try different offsets to induce a loop-abort fault. The intervals should be less than the time for a single instruction. Step 3: Adjust the pulse intensity to optimize the fault rate. Since the fault rate exhibits an unimodal distribution concerning pulse intensity, the peak can be quickly identified using a ternary search method.

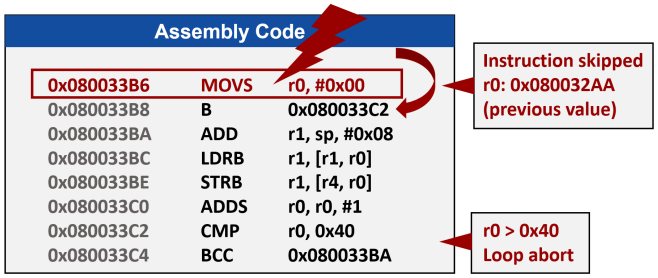


Fig. 6. The mechanism by which instruction skipping causes loop-aborts.

C. Fault Explanation

We determined that the loop-abort faults induced are caused by instruction skipping, which also explains why the time window is close to the execution time of a single instruction.

Figure 6 illustrates the mechanism of loop-abort faults using the assembly code of a loop that assigns values from one array to another. The register $r0$ acts as the loop counter, initialized as zero by the `MOVS` instruction. Then, a `CMP` instruction compares the loop counter with the upper bound (0x40 in the example). The `BCC` instruction jumps to the loop body if the counter is less than the upper bound. Otherwise, the loop terminates. The fault skips the MOVS instruction, and $r0$ remains the previous value, which may exceed the limit (e.g., an address), causing the loop to terminate without executing.

We validated this mechanism through experiments. We used UART to output the value of the loop counter and found that it was a faulty large value when the fault occurred. Additionally, we confirmed that the faults induced are indeed loop-abort

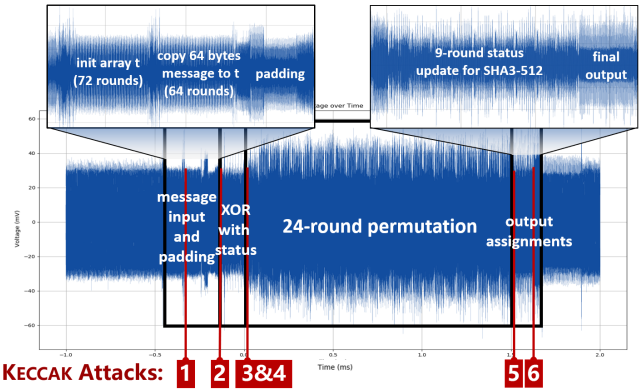


Fig. 7. The side-channel trace of the target SHA3-512 instance for ML-KEM Attack 1 on the STM32F407ZGT6 DUT.

faults. We set and reset a GPIO signal before and after the loop and observed that when the fault occurred, the target array remained at its initial value, and the duration of the GPIO high signal was much shorter than normal.

D. Fault Triggering

Our fault injection experiments are conducted under the condition of inserting triggers before the loop, which can be achieved using the PIFER framework proposed by Qu et al [29]. It allows for modifying the binary and inserting triggers at desired locations on ARM Cortex-M decives.

Additionally, we can leverage side-channel traces to assist in locating the target KECCAK instance and fault injection points. We used a near-field EM probe to measure the side-channel traces of the target SHA3-512 instance for ML-KEM Attack 1 on the STM32F407ZGT6 DUT. As shown in Figure 7, the distinctive features of KECCAK can be observed. We labeled the corresponding operations for each part of the trace and the target positions for the KECCAK attacks.

E. Validation of the Attack Scheme

Above, we have proved the practicality of loop-abort faults on ARM Cortex-M devices (Layer 0). Next, we validate our Layer 1 attacks targeting KECCAK and our Layer 2 attacks targeting ML-KEM and ML-DSA. Our experiments are conducted on the STM32F407ZGT6 DUT using EMFI. We fixed the EM probe's spatial location at the optimal position determined in the characterization (see Figure 3d). Since the fault rate is more sensitive to time and pulse intensity, and we found slight differences in the optimal parameters corresponding to different loops and contexts, we fine-tuned these two parameters to achieve a higher success rate.

1) Validation of the KECCAK Attacks: We targeted the SHA3-512 implementation from the PQClean library and validated the Layer 1 KECCAK attacks. We used a 64-byte input for the function, same as the case of ML-KEM Attack 1 and ML-DSA Attack 1. We performed 1000 fault injection attempts for each parameter setting. The maximum fault rates and the corresponding parameter sets are shown in Table IV. Our six attacks successfully obtained the expected faulty outputs, with success rates ranging from 12.4% to 65.1%. The optimal time offset and pulse intensity are not significantly different, proving that our loop-abort faults are highly reproducible.

TABLE IV
FAULT RATES AND OPTIMAL PARAMETERS OF THE KECCAK ATTACKS

KECCAK Attack	Fault Rate	Offset(ns)	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

TABLE V
FAULT RATES AND OPTIMAL PARAMETERS OF THE ML-KEM AND ML-DSA ATTACKS

Algorithm and Phase	Name of Attack	Fault Rate	Offset (ns)	Intensity(V)
ML-KEM.KeyGen	ML-KEM Attack 1	22.9%	665	260
ML-KEM.Encaps	ML-KEM Attack 3	46.0%	660	270
ML-KEM.Decaps	ML-KEM Attack 6	11.6%	660	270
ML-DSA.KeyGen	ML-DSA Attack 1	18.5%	665	270
ML-DSA.Sign	ML-DSA Attack 3	29.9%	660	260
ML-DSA.Verify	ML-DSA Attack 4	25.1%	665	280

2) Validation of the ML-KEM and ML-DSA Attacks: In our attack scheme, successfully attacking KECCAK and controlling its output is approximately equivalent to successfully performing our Layer 2 attacks. The attacker only needs to target different KECCAK instances in ML-KEM and ML-DSA. For the completeness of the practical attacks, we provide experimental validation of the Layer 2 attacks below.

We targeted the ML-KEM-512 and ML-DSA-44 implementations of the PQClean library on the STM32F407ZGT6 DUT.

First, we selected an attack for each phase and conducted practical fault injection experiments to demonstrate that our scheme applies to all phases of ML-KEM and ML-DSA. We used KECCAK Attack 2 as the Layer 1 attack for these experiments. We also performed 1000 fault injection attempts for each parameter setting to determine the optimal offset and pulse intensity. As shown in Table V [3], our attacks successfully obtained the expected faulty outputs, with success rates ranging from 11.6% to 46.0%. Once the fault is successfully injected, we can achieve key recovery or signature forgery with a 100% probability.

We also validated the multi-point fault attacks ML-KEM Attack 2 (2 faults) and ML-DSA Attack 2 (4 faults). As shown in Table VI, the success rates are 6.2% and 0.9%, respectively. We found that consecutive faults are more likely to cause crashes or resets, resulting in a lower success rate. Table VI also shows the average interval between consecutive faults, a few milliseconds sufficient for ChipSHOUTER to charge. Despite the feasibility of these attacks, we recommend single-point attacks with a higher success rate for fault injection.

[3]For ML-KEM Attack 6, we assume that we can successfully enter the implicit rejection branch, as any fault during the re-encryption can easily achieve this. We focus only on the KECCAK that generates the rejection value.

TABLE VI
FAULT RATES FOR MULTI-POINT FAULT ATTACKS

Name of Attack	No. Faults	Fault Rate	Interval (μs)
ML-KEM Attack 2	2	6.2%	2563.75
ML-DSA Attack 2	4	0.9%	10142.69

TABLE VII
OVERHEAD OF COUNTERMEASURES

Name of Countermeasure	Code Size (Bytes)	Stack Size (Bytes)	Execution Time (Cycles)
None (baseline)	15528	1392	30187
Redundancy	15608, +0.52%	1520, +9.20%	61135, +103%
Loop Counter	16104, +3.71%	1456, +4.60%	30259, +0.24%
Blacklist	16916, +8.94%	1824, +31.0%	30375, +0.62%
Time Checking	15576, +0.31%	1392, +0.00%	30209, +0.07%
Loop Unrolling	54712, +252%	1376, -1.15%	29493, -2.30%

For other attacks, including combinations of all Layer 2 strategies and Layer 1 attacks mentioned in section V, we conducted simulation experiments demonstrating that once a fault occurs, the attack is successful.

VII. COUNTERMEASURES

A. Discussion of Existing Countermeasures

Because KECCAK is an interesting target for side-channel attacks, researchers have used shuffling and masking as countermeasures [17], [30]. However, the shuffling and masking in the literature mainly focus on the KECCAK-f permutation process [31], [32], while our attack targets the higher-level control flow, which cannot be effectively mitigated.

The "Verify after sign" countermeasure checks for invalid signatures but is ineffective against $\mathbf{y}$ attacks (e.g., ML-DSA Attack 3) [10], [17]. The (dynamic) loop counter that checks the number of loop executions is an effective countermeasure against loop-abort faults [6], [17]. It needs to be applied to all vulnerable loops. Redundancy is a general countermeasure and is also used to protect ML-KEM and ML-DSA [9], [12]. It performs multiple calculations and compares their results, increasing the number of faults needed multiple times.

B. Targeted Countermeasures and Evaluation

Blacklist: Because some KECCAK attacks set the output to a fixed pre-known value, these values can be blacklisted and checked against the output. **Time Checking:** When a loop-abort fault occurs, the execution time of KECCAK will be shorter, so its execution time can be checked for fixed-length input/output instances of KECCAK. **Loop Unrolling:** Replace loops with code repetition to avoid the proposed attacks.

We implemented SHA3-512 instances enhanced by the effective countermeasures on the STM32F407 DUT using GCC 7.3.0 compiler with -O0 optimization. As shown in Table VII, we evaluated the time (cycles of execution) and space (code size and stack size) overhead. The redundancy countermeasure doubles the execution time, so it is not recommended for practical use. Though loop unrolling makes loop-abort impossible, it significantly increases the code size. The other three countermeasures do not introduce significant overhead.

VIII. CONCLUSION AND FUTURE WORK

This article demonstrates that manipulating control flow to recover KECCAK outputs enables various novel key recovery, signature forgery, and verification bypass attacks on practical ML-KEM and ML-DSA implementations. The loop-abort fault required for the attacks can be triggered on multiple series of ARM Cortex-M devices, with a success rate of up to 89.5%. Once the fault is successfully injected, the attacks can be carried out with a 100% success probability. For future works, we plan to explore the potential of KECCAK attacks to compromise the security of other post-quantum cryptographic algorithms, such as BIKE [33] and HQC [34].

REFERENCES

[1] J .Bos, L. Ducas, E. Kiltz, et al., "CRYSTALS-Kyber: a CCA-secure module-lattice-based KEM," *2018 EuroS&P*. IEEE, 2018: 353-367.

[2] L. Ducas et al., "CRYSTALS-Dilithium: a Lattice-Based digital signature scheme," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pp. 238–268, Feb. 2018, doi: 10.46586/tches.v2018.i1.238-268.

[3] "Module-Lattice-Based Key-Encapsulation Mechanism Standard," Aug. 2024. doi: 10.6028/nist.fips.203.

[4] "Module-Lattice-Based Digital Signature Standard," Aug. 2024. doi: 10.6028/nist.fips.204.

[5] A. Langlois and D. Stehle, "Worst-case to average-case reductions for module lattices," *Designs Codes and Cryptography*, vol. 75, no. 3, pp. 565–599, Feb. 2014, doi: 10.1007/s10623-014-9938-4.

[6] T. Espitau, P. -A. Fouque, B. Gérard and M. Tibouchi, "Loop-Abort Faults on Lattice-Based Signature Schemes and Key Exchange Protocols," in *IEEE Transactions on Computers*, vol. 67, no. 11, pp. 1535-1549, 1 Nov. 2018, doi: 10.1109/TC.2018.2833119.

[7] P. Ravi, D. B. Roy, S. Bhasin, A. Chattopadhyay, and D. Mukhopadhyay, "Number 'Not Used' Once - Practical Fault Attack on pqm4 Implementations of NIST Candidates," in *Lecture notes in computer science*, 2019, pp. 232–250. doi: 10.1007/978-3-030-16350-1_13.

[8] P. Ravi, M. P. Jhanwar, J. Howe, A. Chattopadhyay, and S. Bhasin, "Exploiting Determinism in Lattice-based Signatures," *Asia CCS 19*, pp. 427–440, Jul. 2019, doi: 10.1145/3321705.3329821.

[9] V. Q. Ulitzsch, S. Marzougui, A. Bagia, M. Tibouchi, and J.-P. Seifert, "Loop Aborts Strike Back: Defeating Fault Countermeasures in Lattice Signatures with ILP," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pp. 367–392, Aug. 2023.

[10] L. Groot Bruinderink and P. Pessl, "Differential Fault Attacks on Deterministic Lattice Signatures," in *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pp. 21–43, Aug. 2018, doi: https://doi.org/10.46586/tches.v2018.i3.21-43.

[11] K. Xagawa, A. Ito, R. Ueno, et al., "Fault-injection attacks against NIST's post-quantum cryptography round 3 KEM candidates," in *Advances in Cryptology-ASIACRYPT 2021*, 2021, pp. 33–61.

[12] P. Pessl and L. Prokop, "Fault attacks on CCA-secure Lattice KEMs," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pp. 37–60, Feb. 2021, doi: 10.46586/tches.v2021.i2.37-60.

[13] J. Hermelink, P. Pessl, and T. Pöppelmann, "Fault-Enabled Chosen-Ciphertext attacks on Kyber," *INDOCRYPT 2021*, 2021, pp. 311–334.

[14] J. Delvaux, and S.M. D. Pozo. "Roulette: Breaking Kyber with Diverse Fault Injection Setups." *IACR Cryptol. ePrint Arch*, 2021.

[15] S. Islam, K. Mus, R. Singh, P. Schaumont and B. Sunar, "Signature Correction Attack on Dilithium Signature Scheme," *2022 EuroS&P*, Genoa, Italy, 2022, pp. 647-663, doi: 10.1109/EuroSP53844.2022.00046.

[16] P. Ravi, B. Yang, S. Bhasin, F. Zhang, and A. Chattopadhyay, "Fiddling the Twiddle Constants - Fault Injection Analysis of the Number Theoretic Transform," in *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pp. 447–481, Mar. 2023.

[17] P. Ravi, A. Chattopadhyay, J. P. D'Anvers, and A. Baksi, "Side-channel and Fault-injection attacks over Lattice-based Post-quantum Schemes (Kyber, Dilithium): Survey and New Results," in *ACM Transactions on Embedded Computing Systems*, vol. 23, no. 2, pp. 1–54, Jun. 2023.

[18] M. J. Kannwischer, P. Pessl, and R. Primas, "Single-Trace attacks on Keccak," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pp. 243–268, Jun. 2020, doi: 10.46586/tches.v2020.i3.243-268.

[19] J. Kelsey, S. Change, and R. Perlner, "SHA-3 derived functions: cSHAKE, KMAC, TupleHash and ParallelHash," Dec. 2016, doi: https://doi.org/10.6028/nist.sp.800-185.

[20] E. Fujisaki and T. Okamoto, "Secure Integration of Asymmetric and Symmetric Encryption Schemes," in *Journal of Cryptology*, vol. 26, no. 1, pp. 80–101, Dec. 2011.
[21] V. Lyubashevsky, "Fiat-Shamir with Aborts: Applications to Lattice and Factoring-Based Signatures," in *ASIACRYPT 2009*, pp. 598–616, 2009, doi: https://doi.org/10.1007/978-3-642-10366-7_35.
[22] N. Bagheri, N. Ghaedi, and S. K. Sanadhya,"Differential fault analysis of SHA-3," in *Progress in Cryptology – INDOCRYPT 2015*, 2015, pp. 253–269. doi: 10.1007/978-3-319-26617-6_14.
[23] P. Luo, Y. Fei, L. Zhang, and A. A. Ding, "Differential Fault Analysis of SHA3-224 and SHA3-256," in *2016 Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*, pp. 4–15, Aug. 2016.
[24] P. Luo, K. Athanasiou, Y. Fei, and T. Wahl, "Algebraic fault analysis of SHA-3," in *Design, Automation &Amp; Test in Europe Conference &Amp; Exhibition (DATE)*, 2015, Mar. 2017, doi: 10.23919/date.2017.7926974.
[25] P. Luo, K. Athanasiou, Y. Fei, and T. Wahl, "Algebraic fault analysis of SHA-3 under relaxed fault models," in *IEEE Transactions on Information Forensics and Security*, vol. 13, no. 7, pp. 1752–1761, Jan. 2018.
[26] J. Bos et al., "CRYSTALS - Kyber: A CCA-Secure Module-Lattice-Based KEM," *2018 EuroS&P*, Apr. 2018.
[27] P. Ravi, M. P. Jhanwar, J. Howe, et al., "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*, pp. 427–440, 2019
[28] N. Bindel, J. Buchmann, and J. Krämer, "Lattice-based signature schemes and their sensitivity to fault attacks," in *2016 Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*, pp. 63–77, 2016.
[29] S. Qu, X. Zhang, C. Zhang, and D. Gu, "Trapped by Your WORDs: (Ab)using Processor Exception for Generic Binary Instrumentation on Bare-metal Embedded Devices," *DAC '24: Proceedings of the 61st ACM/IEEE Design Automation Conference*, pp. 1–6, Jun. 2024.
[30] M. J. Kannwischer, P. Pessl, and R. Primas, "Single-Trace attacks on Keccak," in *DOAJ (DOAJ: Directory of Open Access Journals)*, Jun. 2020, doi: 10.13154/tches.v2020.i3.243-268.
[31] J. Daemen, "Changing of the Guards: a simple and efficient method for achieving uniformity in threshold sharing," in *Lecture notes in computer science*, 2017, pp. 137–153. doi: 10.1007/978-3-319-66787-4_7.
[32] H. Gross, D. Schaffenrath and S. Mangard, "Higher-Order Side-Channel Protected Implementations of KECCAK," in *2017 Euromicro Conference on Digital System Design (DSD)*, Vienna, Austria, 2017, pp. 205-212.
[33] N. Aragon, P. Barreto, S. Bettaieb, et al., "BIKE: bit flipping key encapsulation," 2022.
[34] C. A. Melchor, N. Aragon, S. Bettaieb, et al., "Hamming quasi-cyclic (HQC)," *NIST PQC Round 2*, vol. 4, no. 13, 2018.

Yuxuan Wang received the B.S. degree from Shanghai Jiao Tong University. He is currently pursuing the Ph.D. degree with the School of Electronic Information and Electrical Engineering, Shanghai Jiao Tong University, Shanghai, China. His research interests include side-channel analysis, fault analysis, and post-quantum cryptography.

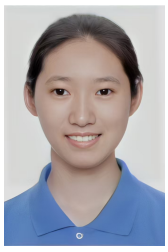
Jintong Yu received the B.S. degree from Harbin Institute of Technology. She is currently pursuing the Ph.D. degree with the School of Electronic Information and Electrical Engineering, Shanghai Jiao Tong University, Shanghai, China. Her research interests include deep-learning based side-channel analysis.

Shipei Qu received the B.S. degree from Xidian University of China, Xi'an, China. He is currently pursuing the Ph.D. degree with the School of Electronic Information and Electrical Engineering, Shanghai Jiao Tong University, Shanghai, China. His research interests include fault analysis, side-channel analysis, and binary analysis.

Xiaolin Zhang received the B.S. degree in cyber security from Xidian University of China, Xi'an, China, in 2020. He is currently pursuing the Ph.D. degree under Successive Postgraduate and Doctoral Program with the School of Electronic Information and Electrical Engineering, Shanghai Jiao Tong University, Shanghai, China. His research interests include design of PUF-based security schemes and cryptography.

Chi Zhang received the B.S. degree from Southeast University and the Ph.D. degree from Shanghai Jiao Tong University. He is currently an Assistant Research Fellow with the School of Electronic Information and Electrical Engineering, Shanghai Jiao Tong University, Shanghai, China. His research interests include side-channel analysis and cryptographic enginerring.

Dawu Gu (Member, IEEE) received the B.S. degree in applied mathematics from Xidian University, Xi'an, China, in 1992, and the M.S. and Ph.D. degrees in cryptography in 1995 and 1998, respectively. He is a Distinguished Professor with the School of Electronic Information and Electrical Engineering, Shanghai Jiao Tong University (SJTU), Shanghai, China. He is the Vice President of the Chinese Association for Cryptologic Research, China. He has got over 200 scientific papers in top academic journals and conferences, such as CRYPTO, IEEE SECURITY AND PRIVACY, ACM CCS, NDSS, and TCHES. His research interests include crypto algorithms, crypto engineering, and system security. Dr. Gu was the winner of Chang Jiang Scholars Distinguished Professors Program in 2014 by Ministry of Education of China. He won the National Award of Science and Technology Progress in 2017. He also served as PC Member of international conferences for more than 30 times.



Xiaowei Li received the B.S. degree from Shanghai Jiao Tong University. She is currently pursuing the Ph.D. degree with the School of Electronic Information and Electrical Engineering, Shanghai Jiao Tong University, Shanghai, China. Her research interests include side-channel analysis and privacy computing.