Keyber: Recovering the Signal by Amplifying the Noise in ML-KEM

Vipin Kumar[1], Prathamesh Ram[1], Suparna Kundu[2] and Dhiman Saha[1]

[1] de.ci.phe.red Lab
Department of Computer Science and Engineering
Indian Institute of Technology Bhilai, India
{vipink,rprathamesh,dhiman}@iitbhilai.ac.in
[2] COSIC, KU Leuven, Leuven, Belgium
suparna.kundu@esat.kuleuven.be

Abstract. ML-KEM (FIPS 203) is now being deployed in real-world applications to provide security against adversaries with quantum computers. Therefore, the resistance of ML-KEM's implementations to physical attacks requires thorough evaluation. So far, fault attacks on ML-KEM have targeted the decapsulation operation. The key generation operation has received little attention from a fault-attack perspective.
In this work, we present Keyber, the first statistical ineffective fault attack (SIFA) on the key generation of ML-KEM. Keyber is a key-recovery attack that needs only a single fault in key generation. The fault adds a known value to the error vector. Thus, the public key becomes faulty, while the seed of the public matrix remains intact. Every subsequent decapsulation, therefore, carries a known induced error. The adversary encapsulates locally and keeps only ciphertexts whose induced error peaks at exactly one coefficient. A decapsulation failure can then be attributed to that coefficient. Each query yields one linear inequality on the secret key, with no further faults.
We observe that approximately 0.2-27% of encapsulations are usable for different parameter sets of ML-KEM. The full secret key is recovered using up to 30,000 inequalities across all security levels. We also realise the attack for ML-KEM-512 on an ARM Cortex-M4 device using a single over-clock fault. We then recover the entire secret key end-to-end. A single fault in this one-time operation thus compromises the key for its entire lifetime. Protecting ML-KEM, therefore, demands hardening not only decapsulation, but also key generation.

Keywords: CRYSTALS-Kyber · ML-KEM · Key Generation · Statistical Ineffective Fault Attack · Post-Quantum Cryptography · Lattice-Based Cryptography

1 Introduction

The National Institute of Standards and Technology (NIST) concluded the first post-quantum cryptography (PQC) standardisation procedure and announced the first-generation standards for Key-Encapsulation Mechanisms (KEMs) and Digital Signature Algorithms (DSAs). The lattice-based PQC scheme CRYSTALS-Kyber [BDK+18], standardised as ML-KEM [Nat24], is one of the two first-generation KEM standards. The conclusion of the NIST standardisation process accelerated the migration from classical public-key cryptography to post-quantum cryptography. As the standardised schemes shift from theory to deployment, exploring their resistance against physical attacks becomes increasingly essential.

A fault attack [BDL97] is a type of physical attack in which an adversary obtains information about the secret key by observing the behaviour after injecting a fault/error

during the execution of a cryptographic algorithm. One of the earliest realisations of a fault attack was the Differential Fault Attack (DFA) [BS97], which compares the outputs of a correct execution and a fault-induced execution to learn secret-related information. Most prior fault attacks on ML-KEM [PP21, HPP21, Del22, WCCL25, KCS+24] are classified as Statistical Ineffective Fault Attacks (SIFAs). This is because they inject faults that may be ineffective, resulting in successful decapsulations. By observing whether or not decapsulation fails, inequalities involving the long-term secret key can be derived and solved using statistical methods. Commonly used statistical methods include Bayesian probability updating [PP21], Belief Propagation [HPP21], CLT-based Belief Propagation [Del22, WCCL25], and Belief Propagation combined with Lattice Reduction algorithms [HMS+23].

The target of all prior SIFAs on ML-KEM has been the decapsulation operation because it utilises the long-term secret key, which remains the same for multiple executions. In contrast, the key generation algorithm has received comparatively little attention [RRB+19, RYB+23] regarding the possibility of fault attacks. Since key generation runs only once, an injected fault persists across every subsequent encapsulation/decapsulation operation, thereby making it an attractive target for fault attacks.

Contribution In this work, we present Keyber, a fault attack on the key generation of ML-KEM. The name joins *Key* and Kyber. The inserted *e* stands for the error vector $\mathbf{e}$, which is where we inject the fault. The attack needs a single fault, and its effect persists for the lifetime of the key.

We *recover* the secret key (*signal*) by injecting a fault (*amplifying*) into the error vector (*noise*) generated during key generation. The fault therefore corrupts $\mathbf{t}$, while $\mathbf{A}$ (and its seed ρ) remains unaffected. This persistent asymmetry causes every subsequent encapsulation-decapsulation interaction to contain a fault-induced error component. The adversary runs encapsulation locally and keeps only ciphertexts in which this error component peaks at a single coefficient. A decapsulation failure can then be attributed to that coefficient, resulting in a linear inequality in the secret values. While prior fault attacks on decapsulation (SIFAs) need one fault per inequality, Keyber needs only a single fault. Several such inequalities are collected and solved using the optimised solver of [Del22]. Our main contributions are as follows.

1. We present Keyber, the first SIFA on the key generation of ML-KEM. Prior SIFAs inject a fault during decapsulation. Keyber injects a single fault during key generation and exploits the information derived from the decapsulation outcomes.

2. We show how rejection sampling during encapsulation attributes a decapsulation failure to exactly one coefficient. Depending on the parameter set, an encapsulation yields a usable ciphertext in 0.2% to 27% of cases.

3. Since no fault is injected during decapsulation, we can generate an arbitrary number of inequalities from a single fault on key generation.

4. We explain why a fault on the public matrix $\mathbf{A}$ fails. The adversary cannot compute the induced error and hence cannot attribute decapsulation failures to a single coefficient.

5. We validate Keyber across all three parameter sets of ML-KEM, using fault magnitudes that require the fewest bit flips. Full key recovery needs up to 20,000 inequalities for ML-KEM-512, 25,000 for ML-KEM-768, and 30,000 for ML-KEM-1024.

6. We introduce a 1-bit fault in a single coefficient of an error polynomial of ML-KEM-512 running on a Cortex-M4 target. An over-clock produces the required fault form with perfect attribution.

Outline This paper is organised as follows. Section 2 introduces the ML-KEM algorithm and describes existing fault attacks in more detail. Section 3 presents our fault on the error vector and the resulting key-recovery attack. It also explains why a fault on the public matrix fails. In Section 4, we demonstrate key recovery in simulation and evaluate the fault on hardware. In addition to that, we propose possible countermeasures in Section 5. Finally, Section 6 concludes the paper.

2 Preliminaries

FIPS 203 [Nat24] is NIST's official standard for Module-Lattice-Based Key-Encapsulation Mechanisms (ML-KEM), which is based on the CRYSTALS-Kyber [BDK+18] scheme. In Section 2.1, we introduce CRYSTALS-Kyber and Section 2.2 describes the existing fault attacks on ML-KEM.

2.1 CRYSTALS-Kyber

Kyber relies on the hardness of the Module-Learning-With-Errors (MLWE) problem. Internally, it employs a CPA-secure Public-Key Encryption scheme (Section 2.1.1) that, when combined with the Fujisaki-Okamoto (FO) transform [FO99], yields a CCA-secure Key-Encapsulation Mechanism (Section 2.1.2). The algorithms are presented in a simplified form, with some modifications to function and variable names to improve comprehension.

Table 1: Approved parameter sets for ML-KEM

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

Kyber has been defined with three parameter sets: ML-KEM-512, ML-KEM-768 and ML-KEM-1024. The parameters n and q define the quotient polynomial ring $\mathcal{R}_q = \mathbb{Z}_q[x]/(x^n+1)$ that Kyber operates in. The degree of any polynomial is $n = 256$, and its coefficients are reduced modulo $q = 3329$, i.e. they lie in the range $[0, q)$. The rank of the underlying module is denoted by k. The Centered Binomial Distribution (CBD), parametrised by η_1 and η_2, is used to sample vectors and polynomials with small coefficients. The ciphertext components $\mathbf{u}$ and v are compressed to d_u and d_v bits, respectively.

We denote a polynomial, a vector of polynomials and a matrix of polynomials using lowercase, boldface lowercase and boldface uppercase characters, respectively. The i-th coefficient of a polynomial x is denoted by $x[i]$. Polynomials, vectors and matrices, when converted to the NTT domain, are denoted with a hat. The hash functions SHA3-512 and SHA3-256 are represented by G and H, respectively, while J represents the SHAKE256 extendable output function. Also note that we sometimes omit the NTT and NTT^{-1} conversions (although they are always included in algorithm definitions) because this is an optimisation that does not affect the correctness of the expressions.

2.1.1 CPA-Secure Kyber-PKE

The Public-Key Encryption scheme Kyber-PKE is defined by three algorithms: (i) KeyGen, to generate a public-private key pair; (ii) Encrypt, to encrypt plaintexts using the public key; and (iii) Decrypt, to decrypt ciphertexts using the private key.

The KeyGen function (K-PKE.KeyGen in Figure 1) first generates two 32-byte pseudo-random seeds by hashing the input randomness d along with the parameter k. The public

K-PKE.KeyGen(d)

Input: randomness d
Output: private key $\mathsf{sk} = \hat{\mathbf{s}}$
Output: public key $\mathsf{pk} = (\hat{\mathbf{t}}, \rho)$
$(\rho, \sigma) \leftarrow G(d \parallel k)$
$\hat{\mathbf{A}} \in \mathcal{R}_q^{k \times k} \leftarrow \mathsf{SampleMatNTT}(\rho)$
$\mathbf{s} \in \mathcal{R}_q^k \leftarrow \mathsf{SampleVecCBD}_{\eta_1}(\sigma, N)$
$\mathbf{e} \in \mathcal{R}_q^k \leftarrow \mathsf{SampleVecCBD}_{\eta_1}(\sigma, N)$
$\hat{\mathbf{s}} \leftarrow \mathsf{NTT}(\mathbf{s})$
$\hat{\mathbf{e}} \leftarrow \mathsf{NTT}(\mathbf{e})$
$\hat{\mathbf{t}} \in \mathcal{R}_q^k \leftarrow \hat{\mathbf{A}} \circ \hat{\mathbf{s}} + \hat{\mathbf{e}}$
return $\mathsf{sk} = \hat{\mathbf{s}}$ and $\mathsf{pk} = (\hat{\mathbf{t}}, \rho)$

K-KEM.KeyGen()

Output: encapsulation key $\mathsf{ek} = (\hat{\mathbf{t}}, \rho)$
Output: decapsulation key $\mathsf{dk} = (\hat{\mathbf{s}}, \hat{\mathbf{t}}, \rho, z)$
$d, z \xleftarrow{\$} \{0,1\}^{256}$
$(\mathsf{sk}, \mathsf{pk}) \leftarrow \mathsf{K\text{-}PKE.KeyGen}(d)$
return $\mathsf{ek} = \mathsf{pk}$ and $\mathsf{dk} = (\mathsf{sk}, \mathsf{ek}, z)$.

K-PKE.Encrypt(pk, r, m)

Input: public key $\mathsf{pk} = (\hat{\mathbf{t}}, \rho)$
Input: randomness r
Input: plaintext polynomial m
Output: ciphertext $ct = (c_1, c_2)$
$\hat{\mathbf{A}} \in \mathcal{R}_q^{k \times k} \leftarrow \mathsf{SampleMatNTT}(\rho)$
$\mathbf{r} \in \mathcal{R}_q^k \leftarrow \mathsf{SampleVecCBD}_{\eta_1}(r, N)$
$\mathbf{e_1} \in \mathcal{R}_q^k \leftarrow \mathsf{SampleVecCBD}_{\eta_2}(r, N)$
$e_2 \leftarrow \mathsf{SampleCBD}_{\eta_2}(r, N)$
$\mathbf{u} \leftarrow \mathsf{NTT}^{-1}(\hat{\mathbf{A}}^\top \circ \hat{\mathbf{r}}) + \mathbf{e_1}$
$v \leftarrow \mathsf{NTT}^{-1}(\hat{\mathbf{t}}^\top \circ \hat{\mathbf{r}}) + e_2 + \left\lceil \frac{q}{2} \right\rfloor m$
$c_1 \leftarrow \mathsf{Compress}_{d_u}(\mathbf{u})$
$c_2 \leftarrow \mathsf{Compress}_{d_v}(v)$
return $ct = (c_1, c_2)$

K-KEM.Encaps(ek)

Input: encapsulation key $\mathsf{ek} = (\hat{\mathbf{t}}, \rho)$
Output: shared secret key K
Output: ciphertext ct
$\mathsf{pk} \leftarrow \mathsf{ek}$
$m \xleftarrow{\$} \{0,1\}^{256}$
$(K, r) \leftarrow G(m \parallel H(\mathsf{pk}))$
$ct \leftarrow \mathsf{K\text{-}PKE.Encrypt}(\mathsf{pk}, r, m)$
return K and ct

K-PKE.Decrypt(sk, ct)

Input: private key $\mathsf{sk} = \hat{\mathbf{s}}$
Input: compressed ciphertext $ct = (c_1, c_2)$
Output: decrypted polynomial m
$\mathbf{u}' \leftarrow \mathsf{Decompress}_{d_u}(c_1)$
$v' \leftarrow \mathsf{Decompress}_{d_v}(c_2)$
$w \leftarrow v' - \mathsf{NTT}^{-1}(\hat{\mathbf{s}}^\top \circ \mathsf{NTT}(\mathbf{u}'))$
$m = \mathsf{Decode}(w)$
return m

K-KEM.Decaps(dk, ct)

Input: decapsulation key $\mathsf{dk} = (\hat{\mathbf{s}}, \hat{\mathbf{t}}, \rho, z)$
Input: ciphertext ct
Output: shared secret key K'
$\mathsf{sk} \leftarrow \hat{\mathbf{s}}$
$\mathsf{pk} \leftarrow (\hat{\mathbf{t}}, \rho)$
$m' \leftarrow \mathsf{K\text{-}PKE.Decrypt}(\mathsf{sk}, ct)$
$(K', r') \leftarrow G(m' \parallel H(\mathsf{pk}))$
$\bar{K} \leftarrow J(z \parallel ct)$
$ct' \leftarrow \mathsf{K\text{-}PKE.Encrypt}(\mathsf{pk}, r', m')$
if $ct' \neq ct$ **then**
    $K' \leftarrow \bar{K}$
end if
return K'

Figure 1: Overview of the ML-KEM algorithms.

matrix $\mathbf{A}$ is then randomly sampled from $\mathcal{R}_q^{k \times k}$ directly in the NTT domain. However, the secret vectors $\mathbf{s}$ and $\mathbf{e}$ need to be randomly sampled from CBD and are later converted to the NTT domain. Sampling from CBD ensures that the polynomials have small coefficients (between $-\eta$ and η). To ensure that the CBD samples are unique, the nonce N is incremented after each sampled polynomial. Finally, the noisy linear system of the MLWE problem is generated by computing $\mathbf{t} = \mathbf{A}\mathbf{s} + \mathbf{e}$. Thus, the secret values are $\mathbf{s}$ and $\mathbf{e}$, while $\mathbf{A}$ and $\mathbf{t}$ are public. The secret key is simply returned as $\hat{\mathbf{s}}$ since the error term is no longer needed. The public key is returned as $(\hat{\mathbf{t}}, \rho)$ because $\mathbf{A}$ can be deterministically

regenerated from ρ, thereby reducing the size of the public key.

The Encrypt function (K-PKE.Encrypt in Figure 1) regenerates the public matrix $\mathbf{A}$ from the pseudorandom seed ρ (part of the public key pk). It then uses the input randomness r as a seed to sample $\mathbf{r}$, $\mathbf{e_1}$ and e_2 from CBD, incrementing the nonce after each polynomial sample. $\mathbf{r}$ and $\mathbf{e_1}$ are length-k vectors of polynomials, whereas e_2 is a single polynomial. The n-bit plaintext m is input as a binary polynomial and is encoded inside the ciphertext component v. The vector $\mathbf{u}$ enables the intended recipient to decode the message correctly. Finally, the ciphertext is returned after compressing $\mathbf{u}$ and v to d_u and d_v bits, respectively. This lossy compression introduces a small error, which the parameters keep within the decryption error bound.

The Decrypt function (K-PKE.Decrypt in Figure 1) decompresses the ciphertext components and computes $w = v' - \mathbf{s}^\top \mathbf{u}'$. The decoder then converts w to the original binary polynomial m. The correctness of decryption is shown below:

$$\begin{aligned}
w &= v' - \mathbf{s}^\top \mathbf{u}' \\
&= (v + \delta v) - \mathbf{s}^\top (\mathbf{u} + \delta \mathbf{u}) \\
&= (\mathbf{t}^\top \mathbf{r} + e_2 + \left\lceil \frac{q}{2} \right\rfloor m + \delta v) - \mathbf{s}^\top (\mathbf{A}^\top \mathbf{r} + \mathbf{e_1} + \delta \mathbf{u}) \\
&= ((\mathbf{A}\mathbf{s} + \mathbf{e})^\top \mathbf{r} + e_2 + \left\lceil \frac{q}{2} \right\rfloor m + \delta v) - \mathbf{s}^\top (\mathbf{A}^\top \mathbf{r} + \mathbf{e_1} + \delta \mathbf{u}) \\
&= ((\mathbf{A}\mathbf{s})^\top \mathbf{r} + \mathbf{e}^\top \mathbf{r} + e_2 + \left\lceil \frac{q}{2} \right\rfloor m + \delta v) - \mathbf{s}^\top (\mathbf{A}^\top \mathbf{r} + \mathbf{e_1} + \delta \mathbf{u}) \\
&= \mathbf{s}^\top \mathbf{A}^\top \mathbf{r} + \mathbf{e}^\top \mathbf{r} + e_2 + \left\lceil \frac{q}{2} \right\rfloor m + \delta v - \mathbf{s}^\top \mathbf{A}^\top \mathbf{r} - \mathbf{s}^\top \mathbf{e_1} - \mathbf{s}^\top \delta \mathbf{u} \\
w &= (\delta v - \mathbf{s}^\top \delta \mathbf{u}) + (\mathbf{e}^\top \mathbf{r} + e_2 - \mathbf{s}^\top \mathbf{e_1}) + \left\lceil \frac{q}{2} \right\rfloor m. \qquad (1)
\end{aligned}$$

δv and $\delta \mathbf{u}$ are the errors introduced due to lossy compression. The choice of d_u and d_v has been made such that the compression error, $\delta v - \mathbf{s}^\top \delta \mathbf{u}$, stays within the error bounds. All vectors and polynomials in the expression $\mathbf{e}^\top \mathbf{r} + e_2 - \mathbf{s}^\top \mathbf{e_1}$ have been sampled from CBD and hence are small. Thus, it is ensured that the standard decryption error, $e_{std} = (\delta v - \mathbf{s}^\top \delta \mathbf{u}) + (\mathbf{e}^\top \mathbf{r} + e_2 - \mathbf{s}^\top \mathbf{e_1})$, is small and thus, stays within the error bounds.

Therefore, the value of w depends on the term $\left\lceil \frac{q}{2} \right\rfloor m$. Hence, if $m[i] = 0$, this term is 0 and the distribution of $w[i]$ is centered at 0. If $m[i] = 1$, the value of this term is $\left\lceil \frac{q}{2} \right\rfloor$ and the distribution of $w[i]$ is centered there. Thus, in order to ensure correct decoding (and hence, decryption), the magnitude of the standard decryption error e_{std} should be less than $\frac{q}{4}$.

2.1.2 CCA-Secure Kyber-KEM

A Key-Encapsulation Mechanism consists of the following three algorithms: (i) KeyGen (key-generation), to generate an encapsulation and decapsulation key; (ii) Encaps (encapsulation), to derive a secret session key and encapsulate information in the form of a ciphertext; and (iii) Decaps (decapsulation), to decapsulate the ciphertext and derive the same session key. Kyber-KEM uses a variant of the FO transform on Kyber-PKE to achieve CCA-security. After the Decaps function decrypts the ciphertext, a re-encryption step is performed. The resulting ciphertext is then compared with the original ciphertext, and if the equality check fails (known as decapsulation failure), the Decaps function returns a garbage value.

The decoder in Line 4 of K-KEM.Decapsin Figure 1 converts each coefficient (a member of $\mathbb{Z}_q$) to a bit, so Decrypt is a many-to-one function. Hence, a large number of malformed ciphertexts exist, which, when queried to a Decrypt oracle (in a chosen ciphertext attack model), may leak information on the long-term secret $\mathbf{s}$. Thus, the re-encryption step prevents adversaries from leveraging malformed ciphertexts.

The KeyGen function (K-KEM.KeyGenin Figure 1) needs no input and starts by generating two 32-byte random values. The first value, d, is passed as input randomness to K-PKE.KeyGen. The second value z becomes a part of the decapsulation key and will be used to generate the garbage value that Decaps returns in case of a decapsulation failure. The private-public key pair returned by K-PKE.KeyGen is used to form the encapsulation and decapsulation keys of the Key-Encapsulation Mechanism.

The Encaps function (K-KEM.Encapsin Figure 1) is also straightforward and starts by generating a random 32-byte plaintext m. This is then concatenated with the public key hash and further hashed to obtain two 32-byte values, K and r. K is the secret session key, while r is passed as input randomness to K-PKE.Encrypt. The Encaps function returns the secret session key K and the ciphertext ct. The ciphertext is then sent for decapsulation.

The Decaps function (K-KEM.Decapsin Figure 1) first decrypts the ciphertext to get the decrypted plaintext polynomial m'. Then, the re-encryption process begins by regenerating the randomness r' and the secret session key K'. A garbage value for K' is computed using z, to be used in case of decapsulation failure. The decrypted plaintext is then re-encrypted to get ct', which is compared with the original ciphertext ct. If they match, the secret session key is returned; otherwise, the garbage value is returned.

2.2 Existing Fault-Injection Attacks on ML-KEM

The first fault-injection attack on ML-KEM [RRB+19] targets the KeyGen function. The fault is injected into the nonce, preventing the increment, thereby forcing nonce reuse. The vectors $\mathbf{s}$ and $\mathbf{e}$ are then sampled identically, so $\mathbf{e} = \mathbf{s}$ and

$$\mathbf{t} = \mathbf{A}\mathbf{s} + \mathbf{e} = (\mathbf{A} + 1)\,\mathbf{s}.$$

This is a very weak instance of the MLWE problem, which can be solved by Gaussian elimination. The same fault during encapsulation makes $\mathbf{r} = \mathbf{e_1}$.

Another fault attack [RYB+23] that targets the KeyGen function uses a stuck-at-0 fault model to zeroize all twiddle factors used within NTT. This results in a huge reduction in the entropy of the secret variables involved in KeyGen and Encaps. This greatly reduces the brute-force complexity, thereby enabling efficient recovery of the secret key.

2.2.1 Statistical Ineffective Fault Attacks on ML-KEM

We now review existing fault attacks targeting the Decaps function, all of which are classified as Statistical Ineffective Fault Attacks. Subsequently, Section 2.2.2 identifies the common underlying principle that enables these attacks to achieve successful secret-key recovery.

Message Decoder Fault Attack The earliest fault attack targeting Decaps was by Pessl and Prokop [PP21]. They target a side-channel-resistant version of the message decoder (Line 4 of K-PKE.Decrypt). In order to avoid timing-based attacks, a constant-time implementation of the message decoder is used (for each coefficient of the polynomials).

$$m = \left(\left(2w + \frac{q}{2}\right) \;/\; q\right) \;\&\; 1$$

Through an instruction-skip fault, they skip the addition by $\frac{q}{2}$ for a single coefficient of their choosing. This has the following effect:

$$\begin{aligned} m &= ((2w) \;/\; q) \;\&\; 1 \\ &= \left(\left(2w - \frac{q}{2} + \frac{q}{2}\right) \;/\; q\right) \;\&\; 1 \\ &= \left(\left(2\left(w - \frac{q}{4}\right) + \frac{q}{2}\right) \;/\; q\right) \;\&\; 1 \end{aligned}$$

As a result of the instruction skip, that coefficient decodes $(w - \frac{q}{4})$ to m. This greatly increases the likelihood of decoding to an incorrect m, resulting in an incorrect re-encrypted ciphertext and ultimately leading to decapsulation failure. By observing whether each decapsulation succeeds or fails, they derive an inequality involving the secret key. Once enough inequalities have been generated, they recover the secret key by solving the system of inequalities using a statistical approach.

Ciphertext Correction Fault Attack Hermelink *et al.* [HPP21] target the Decaps function using a different approach. Rather than querying an honestly encapsulated ciphertext ct, they add $\frac{q}{4}$ to any one coefficient and query this malformed ciphertext ct^*. Thus, for that coefficient, $(w + \frac{q}{4})$ is decoded to m. They inject a fault just before the ciphertext comparison and attempt to equate the ciphertexts to pass the equality check. If the decoder had returned the original m, the re-encryption would have returned ct. It is thus possible to convert ct to ct^* or vice versa via fault injection, resulting in a successful decapsulation. However, if the decoder returns an incorrect m, the re-encryption yields a completely different ciphertext, since the encryption randomness depends on m. Thus, the injection of a simple fault cannot make the ciphertexts equal, resulting in a decapsulation failure. Thus, they derive inequalities involving the secret key and solve the system of inequalities using belief propagation.

Roulette Family of Fault Attacks [Del22] diversifies the attack by Hermelink *et al.* in the following four ways. First, they expand the attack surface by introducing additional locations for fault injection. Second, they relax the fault model by supporting set-to-0 faults, random faults, arbitrary bit flips, and instruction skips. Third, they demonstrate how masking and blinding methods do not hinder, but rather help their attack. Fourth, they propose a more efficient and error-tolerant inequality solver.

Masked Message Decoder Fault Attack Wang *et al.* [WCCL25] target a masked implementation of the message decoder. They show that masking simplifies fault injection, since its inherent randomness causes stuck-at-0 faults to be converted into bit flips with some probability. Since the masked message decoder is implemented using bitwise operations, a single bit flip can cause decapsulation failure, provided the error remains within a specific bound. Thus, inequalities involving the secret key are derived and solved using a Central Limit Theorem (CLT)-based statistical solver.

2.2.2 Underlying Principle of SIFAs on ML-KEM

Parameters for ML-KEM are chosen such that the standard error e_{std} rarely exceeds the error bound. Hence, without fault injection, there is only a negligible chance of decapsulation failure. Although a successful decapsulation yields inequalities of the form $|e_{std}| < \frac{q}{4}$, they provide limited distinguishing information, as the inequalities are satisfied not only by the correct secret key but also by most candidate solutions. When used in isolation, successful decapsulations introduce another limitation: low-Hamming-weight candidates for $\mathbf{s}$ are never eliminated (see Section 3.2 for more details). Hence, successful secret key recovery requires highly distinguishing inequalities arising from both successful and failed decapsulations.

Thus, SIFAs attempt to introduce an induced error e_{ind}, which may lead to incorrect decoding and decapsulation failures. Each attack uses a different method to inject faults that lead to this induced error before decoding. [PP21] skips an instruction in the decoder to induce an error of $\frac{-q}{4}$. [HPP21, Del22] induce an error of $\frac{q}{4}$ by simply manipulating the ciphertexts. However, they need a corrective fault to ensure that a correct decoding results in a successful decapsulation. An induced error of $\pm\frac{q}{4}$ is favourable because it

yields roughly equal numbers of decapsulation successes and failures. This generates highly distinguishing inequalities that aid in successful secret key recovery. Figure 2 shows the impact of an induced error on the decoding process.

[WCCL25] deviates from this pattern as they directly flip a bit in the masked message decoder. This does not cleanly simplify to the addition of an induced error of a certain magnitude. However, they show that the secret key can be recovered even though their fault causes a decapsulation failure in 99% of the cases.
In Section 3.4, we use the core principle of existing SIFAs on ML-KEM to decide the exact magnitude of our fault.

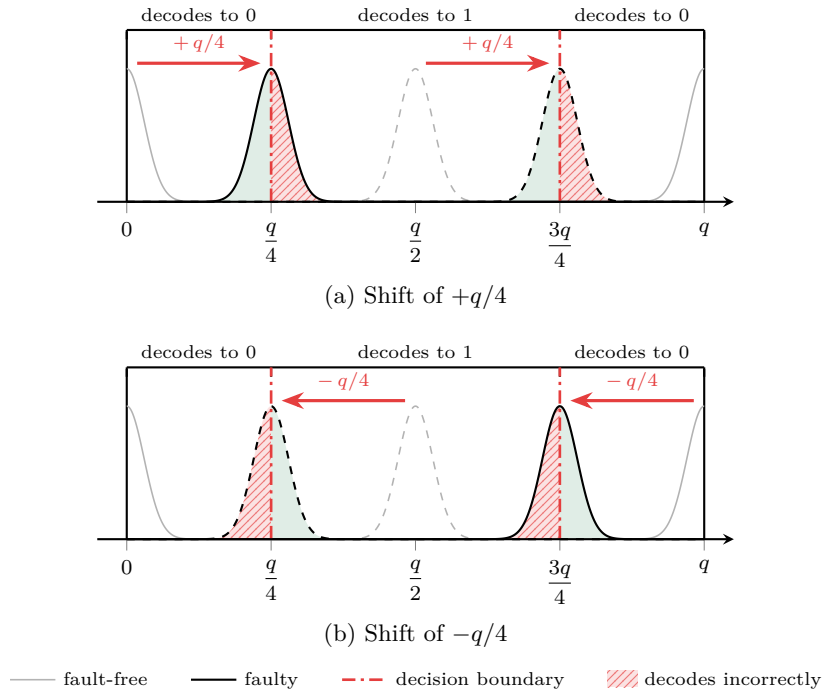


Figure 2: Distribution of $w[i]$ after an induced error of (a) $\frac{q}{4}$ and (b) $-\frac{q}{4}$. Grey curves show the fault-free distribution of $w[i]$. Black curves show the faulty ones. Solid lines represent $m = 0$, and dashed lines represent $m = 1$. Red arrows mark the shift. Red dash-dotted lines mark the decision boundaries. Plain regions shaded in green decode correctly despite the shift. Hatched regions shaded in red cross the boundary and decode incorrectly.

3 Statistical Ineffective Fault Attack on KeyGen

The KeyGen algorithm of ML-KEM is an attractive target for fault injection. It runs only once, so a single fault corrupts the key pair for its entire lifetime. Every later encapsulation and decapsulation then inherits the fault. This persistence gives Keyber a structure that prior SIFAs lack. We inject the fault during KeyGen, but its effect is observed only later, through the success or failure of Decaps. We call KeyGen the injection site and Decaps the observation site. Prior SIFAs used one site for both, namely the Decaps algorithm. Keyber keeps the injection site and the observation site apart.

This section is organised as follows. Section 3.1 states the adversarial model, describes the fault location and explains how it propagates to the observation site. Section 3.2 explains the two requirements for successful key recovery. In Section 3.3, we show that the requirements are satisfied through a rejection sampling process. We decide the exact fault

magnitude in Section 3.4 and Section 3.5 presents the full attack workflow. Section 3.6 concludes this section by explaining why a fault on the public matrix $\mathbf{A}$ would not work.

3.1 Faulting the Error Vector

As outlined in [RCDB24], KEMs can operate in two settings, depending upon the longevity of the key pair: (i) Static-key setting, where a single key pair is used for multiple key establishments, and (ii) Ephemeral-key setting, where a fresh key pair is used for every key establishment.

Adversarial Model Similar to existing SIFAs on ML-KEM, we work in the static-key setting. We assume that the adversary can inject simple bit-flip faults during the execution of KeyGen. The targeted device stores the expanded decapsulation key, so the faulty public key is reused in every decapsulation. Once the faulty public key is published, the adversary can run Encaps locally, since it uses no secret values. They store all intermediate values of each encapsulation, including the session key. The adversary also has access to a decapsulation oracle. They query it with their ciphertexts and compare the returned session key with their own. This tells them whether decapsulation succeeded or failed. Finally, they turn these outcomes into inequalities and solve them for the long-term secret key.

Fault Injection The public key of ML-KEM has two parts. While the vector $\hat{\mathbf{t}}$ is stored in full, the public matrix $\hat{\mathbf{A}}$ is replaced by the 32-byte seed ρ that it is generated from. We inject the fault into the error vector $\mathbf{e}$, right after it is sampled. Algorithm 1 shows the location of the fault. Faulty values are marked with an asterisk. The fault leaves ρ and $\mathbf{A}$ untouched. However, $\hat{\mathbf{t}}$ becomes faulty because it is computed from the faulty $\hat{\mathbf{e}}^*$.

Algorithm 1 K-PKE.Faulty-KeyGen(d)

Input: randomness d
Output: private key $\mathsf{sk} = \hat{\mathbf{s}}$
Output: public key $\mathsf{pk} = (\hat{\mathbf{t}}^*, \rho)$

1: $(\rho, \sigma) \leftarrow G(d \parallel k)$
2: $\hat{\mathbf{A}} \in \mathcal{R}_q^{k \times k} \leftarrow \mathsf{SampleMatNTT}(\rho)$
3: $\mathbf{s} \in \mathcal{R}_q^k \leftarrow \mathsf{SampleVecCBD}_{\eta_1}(\sigma)$
4: $\mathbf{e} \in \mathcal{R}_q^k \leftarrow \mathsf{SampleVecCBD}_{\eta_1}(\sigma)$
5: $\mathbf{e}^* \leftsquigarrow \mathbf{e} + \Delta\mathbf{e}$ ▷ Injecting a fault into $\mathbf{e}$
6: $\hat{\mathbf{s}} \leftarrow \mathsf{NTT}(\mathbf{s})$
7: $\hat{\mathbf{e}}^* \leftarrow \mathsf{NTT}(\mathbf{e}^*)$
8: $\hat{\mathbf{t}}^* \in \mathcal{R}_q^k \leftarrow \hat{\mathbf{A}} \circ \hat{\mathbf{s}} + \hat{\mathbf{e}}^*$ ▷ Faulty $\hat{\mathbf{e}}^*$ results in a faulty $\hat{\mathbf{t}}^*$
9: **return** $\mathsf{sk} = \hat{\mathbf{s}}$ and $\mathsf{pk} = (\hat{\mathbf{t}}^*, \rho)$

Fault Propagation The vector $\mathbf{e}$ is used only inside KeyGen and is never regenerated. The fault therefore reaches Encaps and Decaps only through $\hat{\mathbf{t}}^*$. The first ciphertext component stays fault-free as it does not use $\mathbf{t}$, and is computed as $\mathbf{u} = \mathbf{A}^\top\mathbf{r} + \mathbf{e_1}$. Although Encaps (Line 3, K-KEM.Encaps) derives K and r from a hash of the faulty public key, we do not consider them to be faulty. This is because the same faulty key is hashed to regenerate K and r during the re-encryption stage of Decaps (Line 4, K-KEM.Decaps). Since both algorithms derive identical values of K and r, we treat them as fault-free. The second

ciphertext component changes as follows:

$$\begin{aligned} v^* &= \mathbf{t}^{*\top}\mathbf{r} + e_2 + \left\lceil \frac{q}{2} \right\rfloor m \\ &= (\mathbf{A}\mathbf{s} + \mathbf{e}^*)^\top \mathbf{r} + e_2 + \left\lceil \frac{q}{2} \right\rfloor m \\ &= \mathbf{s}^\top \mathbf{A}^\top \mathbf{r} + \mathbf{e}^{*\top}\mathbf{r} + e_2 + \left\lceil \frac{q}{2} \right\rfloor m \end{aligned}$$

Decryption inside Decaps then computes

$$\begin{aligned} w &= v^{*\prime} - \mathbf{s}^\top \mathbf{u}' \\ &= (v^* + \delta v) - \mathbf{s}^\top(\mathbf{u} + \delta\mathbf{u}) \\ &= \mathbf{s}^\top \mathbf{A}^\top \mathbf{r} + \mathbf{e}^{*\top}\mathbf{r} + e_2 + \left\lceil \frac{q}{2} \right\rfloor m + \delta v - \mathbf{s}^\top \mathbf{A}^\top \mathbf{r} - \mathbf{s}^\top \mathbf{e_1} - \mathbf{s}^\top \delta\mathbf{u} \\ &= (\mathbf{e} + \Delta\mathbf{e})^\top \mathbf{r} + e_2 + \left\lceil \frac{q}{2} \right\rfloor m + \delta v - \mathbf{s}^\top \mathbf{e_1} - \mathbf{s}^\top \delta\mathbf{u} \\ w &= \Delta\mathbf{e}^\top \mathbf{r} + (\delta v - \mathbf{s}^\top \delta\mathbf{u}) + (\mathbf{e}^\top \mathbf{r} + e_2 - \mathbf{s}^\top \mathbf{e_1}) + \left\lceil \frac{q}{2} \right\rfloor m. \end{aligned} \tag{2}$$

Compared to Equation (1), Equation (2) has one extra term. We call it the induced error, $e_{ind} = \Delta\mathbf{e}^\top\mathbf{r}$. The adversary knows e_{ind} exactly, since they choose $\Delta\mathbf{e}$ and generate $\mathbf{r}$ during Encaps. The total error is the sum of the induced error and the standard error e_{std} of Equation (1). We regroup it as

$$\begin{aligned} e_{tot} &= e_{ind} + e_{std} \\ &= \Delta\mathbf{e}^\top\mathbf{r} + \mathbf{s}^\top(-\mathbf{e_1} - \delta\mathbf{u}) + \mathbf{e}^\top\mathbf{r} + (e_2 + \delta v). \end{aligned} \tag{3}$$

The only unknowns in Equation (3) are $\mathbf{s}$ and $\mathbf{e}$, and they appear linearly. Decapsulation succeeds if every coefficient of e_{tot} stays within $(-\frac{q}{4}, \frac{q}{4})$ and fails otherwise. As shown in Figure 2, an induced error of $\pm\frac{q}{4}$ centers the distribution of $w[k]$ onto a decision boundary. Without the fault, $|e_{std}[k]| < \frac{q}{4}$ almost always holds, so every query succeeds and reveals nothing. With the fault, decapsulation success or failure dictates the following inequalities.

$$|e_{ind}[k] + e_{std}[k]| \begin{cases} < \frac{q}{4} & \text{if Decaps succeeds} \\ > \frac{q}{4} & \text{if Decaps fails} \end{cases} \tag{4}$$

Thus, the outcome of a query yields one inequality on the secret. Inequality generation is explained in detail in Section 4.1.

3.2 Requirements for Key Recovery

A decapsulation outcome becomes useful only if two conditions hold.

Attribution Equation (3) holds for each of the 256 coefficients. A successful decapsulation shows that all 256 coefficients stayed within the bound. A failure only shows that at least one coefficient crossed it. To write an inequality on a failure, the adversary must know which coefficient caused it. Existing SIFAs have this property by design. Since they inject faults in Decaps, they can limit the effect of their fault to exactly one coefficient of w, ensuring that the other 255 coefficients stay far from the error bound.

Failure Inequalities Successful decapsulations give inequalities of the form $e_{tot} > -\frac{q}{4}$ or $e_{tot} < \frac{q}{4}$, depending on the sign of e_{ind}. However, these inequalities in isolation are not sufficient, as a large set of small candidates satisfy them. For example, $\mathbf{s} = \mathbf{e} = 0$ keeps e_{std} close to 0. It is unlikely that any number of such inequalities can rule out a majority

of these small candidates. Hence, a solver cannot single out the true key. Failures give inequalities of the form $e_{tot} > \frac{q}{4}$ or $e_{tot} < -\frac{q}{4}$. A combination of both types of inequalities is much more likely to hone in on the correct key.

Failures are only usable with attribution. The rest of this section therefore chooses $\Delta\mathbf{e}$ so that each queried ciphertext has exactly one coefficient that may cause a failure.

3.3 Rejection Sampling: Choosing Favourable Samples of $\mathbf{r}$

The adversary can control $\Delta\mathbf{e}$, but can only observe $\mathbf{r}$, as it is sampled from the Centered Binomial Distribution during Encaps. Since $\mathbf{r}$ has many non-zero coefficients, $e_{ind} = \Delta\mathbf{e}^\top\mathbf{r}$ cannot be confined to a single coefficient. However, for certain (favourable) samples of $\mathbf{r}$, the magnitude of e_{ind} reaches an extremum (maximum or minimum) at exactly one coefficient. This ensures that a decapsulation failure is most likely caused by that coefficient of e_{ind}. Thus, we use a rejection sampling process to keep only those ciphertexts that have been generated from a favourable sample of $\mathbf{r}$. We now show that simple faults on $\mathbf{e}$ are sufficient to ensure that rejection sampling is feasible.

3.3.1 Fault on a Single Coefficient

We start with the simplest fault, $\delta\mathbf{e} = cx^0$, applied to the first polynomial of $\mathbf{e}$.

$$\Delta\mathbf{e} = \begin{bmatrix} \delta\mathbf{e} \\ 0 \\ 0 \end{bmatrix}$$

The induced error is then

$$\begin{aligned} e_{ind} = \Delta\mathbf{e}^\top\mathbf{r} &= \begin{bmatrix} \delta\mathbf{e} \\ 0 \\ 0 \end{bmatrix}^\top \begin{bmatrix} r_0 \\ r_1 \\ r_2 \end{bmatrix} \\ &= \delta\mathbf{e} \cdot r_0 \\ &= cx^0 \cdot (r_0[0] + r_0[1]x + \cdots + r_0[255]x^{255}) \\ &= c\,r_0[0] + c\,r_0[1]x + \cdots + c\,r_0[255]x^{255}. \end{aligned}$$

As $\mathbf{r}$ is sampled from CBD parametrised by η_1, any coefficient $r_0[i]$ can take values in $[-\eta_1, \eta_1]$. Hence, the maximum and minimum value of $e_{ind}[i]$ is $\eta_1 c$ and $-\eta_1 c$, respectively. Naturally, the following two cases arise.

$\eta_1 = 2$ (ML-KEM-768 and ML-KEM-1024) A CBD sample with $\eta_1 = 2$ can take a value in $\{-2, -1, 0, 1, 2\}$ with corresponding probabilities $\{\frac{1}{16}, \frac{4}{16}, \frac{6}{16}, \frac{4}{16}, \frac{1}{16}\}$. Thus, $e_{ind}[i]$ reaches an extremum if $r_0[i]$ is -2 or 2. The probability of this event is

$$\tfrac{1}{16} + \tfrac{1}{16} = \tfrac{2}{16}.$$

Let X be the number of coefficients of e_{ind} that reach an extremum. The expected value of X is

$$\mathbb{E}[X] = 256 \times \tfrac{2}{16} = 32.$$

Since coefficients of e_{ind} are independent from each other, the probability that exactly one coefficient reaches an extremum is

$$\Pr(X = 1) = \binom{256}{1}\left(\tfrac{2}{16}\right)\left(\tfrac{14}{16}\right)^{255} \approx 5.21 \times 10^{-14}.$$

Rejection sampling at this rate is impractical. The same holds for $\delta\mathbf{e} = cx^i$ with any $i \in [0, 255]$.

$\eta_1 = 3$ (ML-KEM-512) A CBD sample with $\eta_1 = 3$ can take a value in $\{-3, -2, -1, 0, 1, 2, 3\}$ with corresponding probabilities $\{\frac{1}{64}, \frac{6}{64}, \frac{15}{64}, \frac{20}{64}, \frac{15}{64}, \frac{6}{64}, \frac{1}{64}\}$. Thus, $e_{ind}[i]$ reaches an extremum if $r_0[i]$ is -3 or 3. The probability of this event is

$$\tfrac{1}{64} + \tfrac{1}{64} = \tfrac{2}{64}.$$

The expected number of coefficients that reach an extremum is given by

$$\mathbb{E}[X] = 256 \times \tfrac{2}{64} = 8.$$

Since coefficients of e_{ind} are independent, the probability that exactly one coefficient reaches an extremum is

$$\Pr(X = 1) = \binom{256}{1} \left(\tfrac{2}{64}\right) \left(\tfrac{62}{64}\right)^{255} \approx 0.002.$$

This means that we expect to get 2 usable ciphertexts for every 1000 encapsulations. Although the probability is quite low, Encaps is a public operation that the adversary runs offline, and the generation is easily parallelised and can be driven well below its naive cost.

Therefore, if we inject a fault on a single coefficient, rejection sampling is feasible only for ML-KEM-512.

3.3.2 Fault on Two Coefficients

We now explore faults on two coefficients of $\mathbf{e}$ to improve rejection sampling probability. The simplest fault is $\delta\mathbf{e} = cx^0 + cx^1$. Thus, the induced error becomes

$$\begin{aligned} e_{ind} = \Delta\mathbf{e}^\top\mathbf{r} &= \delta\mathbf{e} \cdot r_0 \\ &= (cx^0 + cx^1) \cdot (r_0[0] + r_0[1]x + \cdots + r_0[255]x^{255}) \\ &= c(r_0[0] - r_0[255]) + c(r_0[0] + r_0[1])x + \cdots + c(r_0[254] + r_0[255])x^{255}. \end{aligned}$$

Let us say that $e_{ind}[i]$ depends on $r_0[j]$ and $r_0[k]$. As each coefficient of e_{ind} now depends on two coefficients of r_0, its maximum and minimum value is $-2\eta_1 c$ and $2\eta_1 c$, respectively. For this, both coefficients of r_0 must take an extreme value (either $-\eta_1$ or η_1). Their exact signs however, depend on whether the expression for $e_{ind}[i]$ involves an addition or a subtraction.

$\eta_1 = 2$ (ML-KEM-768 and ML-KEM-1024) Regardless of the exact sign of $r_0[j]$ and $r_0[k]$, they each must be 2 or -2 simultaneously and thus, reach either a maximum or a minimum. The probability of this event is

$$\tfrac{1}{16} \times \tfrac{1}{16} + \tfrac{1}{16} \times \tfrac{1}{16} = \tfrac{2}{256}.$$

Let X be the number of coefficients that reach an extremum. The expected value of X is

$$\mathbb{E}[X] = 256 \times \tfrac{2}{256} = 2.$$

We again treat each coefficient of e_{ind} as independent. This is not exact, since each coefficient of r_0 appears in two coefficients of e_{ind}. However, the approximation is still close to the empirical rate and the attack does not depend on the precision of rejection sampling, hence, we proceed. The probability that exactly one coefficient reaches an extremum is

$$\Pr(X = 1) = \binom{256}{1} \left(\tfrac{2}{256}\right) \left(\tfrac{254}{256}\right)^{255} \approx 0.27.$$

Thus, about 1 in 4 encapsulations yield usable ciphertexts. This makes rejection sampling highly practical. The same analysis holds for any two positions, so we use $\delta\mathbf{e} = cx^i + cx^j$ with $i \neq j \in [0, 255]$.

$\eta_1 = 3$ (ML-KEM-512) Regardless of the exact sign of $r_0[j]$ and $r_0[k]$, they each must be 3 or -3 simultaneously and thus, reach either a maximum or a minimum. The probability of this event is

$$\frac{1}{64} \times \frac{1}{64} + \frac{1}{64} \times \frac{1}{64} = \frac{2}{4096}.$$

Let X be the number of coefficients that reach an extremum. The expected value of X is

$$\mathbb{E}[x] = 256 \times \frac{2}{4096} = \frac{1}{8}.$$

Treating each coefficient as independent, the probability that exactly one coefficient reaches an extremum is

$$\Pr(X = 1) = \binom{256}{1} \left(\frac{2}{4096}\right) \left(\frac{4094}{4096}\right)^{255} \approx 0.11.$$

About 1 in 9 encapsulations yield usable ciphertexts. Compared to a single coefficient fault, the rejection sampling probability has increased for $\eta_1 = 3$ as well. However, due to the practicality of the fault, a single coefficient fault proves to be the better attack in the case of ML-KEM-512. We explain this in more detail in Section 3.4 where we fix the magnitude of fault that allows successful key recovery.

b

3.4 Determining the Fault Magnitude

The strategy of existing SIFAs on ML-KEM has been to introduce an error of magnitude $\pm\frac{q}{4}$ into any coefficient of w. This results in roughly equal numbers of decapsulation failures and successes and thus generates inequalities that carry maximal information on the secrets. However, [WCCL25] successfully recover the secret key by injecting a fault that results in a 99% decapsulation failure rate. Hence, we note that the extreme value of $e_{ind}[i]$ need not go up to $\pm\frac{q}{4}$ for successful key recovery as long as decapsulation failures occur reasonably often. We now consider the three cases that admit a feasible rejection sampling process.

Single Coefficient Fault on ML-KEM-512 In this case, the extreme value of $e_{ind}[i]$ is $\pm\eta_1 c = \pm 3c$. The value of c that makes $e_{ind}[i] = \pm\frac{q}{4}$ is

$$\begin{aligned} \pm 3c &= \pm\frac{q}{4} \\ c &= \frac{q}{12} \\ c &\approx 277 = 2^8 + 2^4 + 2^2 + 2^0 \end{aligned}$$

Hence, a fault magnitude of 277 needs 4 bit-flips, which is relatively high. Since we don't need to go up to $\frac{q}{4}$, we can choose a simpler fault magnitude of **256**. In this case, the extremum of $e_{ind}[i]$ is $\pm 3 \times 256 = \pm 768$. As this fault magnitude needs only a single bit-flip in total, the attack becomes much more practical.

Two Coefficient Fault on ML-KEM-512 When a fault in injected into two coefficients of $\mathbf{e}$, the extremum of $e_{ind}[i]$ is $\pm 2\eta_1 c = \pm 6c$. The value of c that makes $e_{ind}[i] = \pm\frac{q}{4}$ is

$$\begin{aligned} \pm 6c &= \pm\frac{q}{4} \\ c &= \frac{q}{24} \\ c &\approx 138 = 2^7 + 2^3 + 2^1 \end{aligned}$$

Hence, a fault magnitude of 138 needs 3 bit-flips in two random coefficients, which is 6 bit-flips in total. Even if we choose a fault magnitude of **128**, where the extremum of $e_{ind}[i]$ is $\pm 6 \times 128 = \pm 768$, we require 2 bit-flips in total (one in each coefficient).

Since the practicality of the fault injection carries more weight than the probability of rejection sampling, we conclude that a single coefficient fault of magnitude 256 is our best attack on ML-KEM-512.

Two Coefficient Fault on ML-KEM-768 and ML-KEM-1024 In this case, the extreme value of $e_{ind}[i]$ is $\pm 2\eta_1 c = \pm 4c$. The value of c that makes $e_{ind}[i] = \pm\frac{q}{4}$ is

$$\begin{aligned} \pm 4c &= \pm\frac{q}{4} \\ c &= \frac{q}{16} \\ c &\approx 208 = 2^7 + 2^6 + 2^4 \end{aligned}$$

Hence, a fault magnitude of **208** needs 3 bit-flips in two random coefficients, which is 6 bit-flips in total. A simpler fault magnitude of **192** results in a total of 4 bit-flips (two in each coefficient).

The chosen fault magnitudes for each parameter set of ML-KEM are summarised in Table 2 along with the number of inequalities needed for successful key recovery. The inequality generation procedure is explained in Section 4.1.

3.5 Attack Workflow

A high-level overview of the attack is shown in Figure 3 while the exact steps are described below.

1. The adversary injects the fault $\Delta\mathbf{e} = \begin{bmatrix} \delta\mathbf{e} \\ 0 \\ 0 \end{bmatrix}$ during KeyGen. For a single coefficient fault, $\delta\mathbf{e} = cx^i$ for any $i \in [0, 255]$. For a two coefficient fault, $\delta\mathbf{e} = cx^i + cx^j$ for some $i \neq j \in [0, 255]$.

2. Once the faulty public key is published, multiple encapsulations are run locally by the adversary.

3. For each encapsulation, the adversary computes $e_{ind} = \Delta\mathbf{e}^\top\mathbf{r}$. The resulting ciphertext is rejected unless exactly one coefficient $e_{ind}[k]$ reaches an extremum. The extremum is $\pm\eta_1 c$ for a single coefficient fault and $\pm 2\eta_1 c$ for a two coefficient fault.

4. Each accepted ciphertext is queried to the decapsulation oracle, which returns a session key.

5. They compare the returned session key with the one generated during Encaps. A match indicates decapsulation success, and a mismatch points to decapsulation failure.

6. Each decapsulation outcome yields one inequality on coefficient k of Equation (3). A success gives $|e_{tot}[k]| < \frac{q}{4}$. A failure gives $|e_{tot}[k]| \geq \frac{q}{4}$. Since $e_{ind}[k]$ is known, each inequality constrains $e_{std}[k]$, which is linear in $\mathbf{s}$ and $\mathbf{e}$.

7. All inequalities are passed to an inequality solver, which returns $\mathbf{s}$ and $\mathbf{e}$.

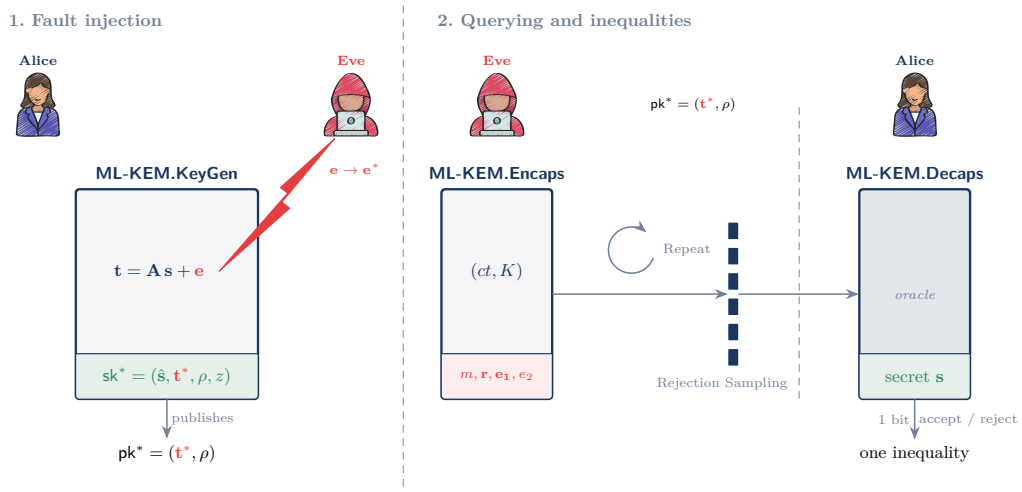


Figure 3: After the adversary injects a fault in KeyGen, Alice publishes her faulty public-key. Then the adversary runs multiple honest encapsulations and uses rejection sampling to select usable ciphertexts. Querying these ciphertexts to a decapsulation oracle yields inequalities that are solved to recover the secret key.

3.6 Why Not Fault the Matrix A?

A fault on $\hat{\mathbf{A}}$ creates the same asymmetry as a fault on $\mathbf{e}$. Suppose the fault hits $\hat{\mathbf{A}}$ after Line 2 of K-PKE.KeyGen. Then $\hat{\mathbf{t}}^*$ is computed from the faulty matrix, while ρ stays fault-free. Encapsulation regenerates the correct $\mathbf{A}$ from ρ. Repeating the derivation of Equation (2) gives the induced error

$$e_{ind} = \mathbf{s}^\top \Delta\mathbf{A}^\top \mathbf{r}. \tag{5}$$

We tried this target first, but it failed for two reasons.

No attribution The induced error in Equation (5) contains the secret $\mathbf{s}$. The adversary can neither compute it nor confine it to one coefficient. Any of the 256 coefficients may then cause a failure, and failures cannot be attributed. Only successful decapsulations remain usable. As Section 3.2 shows, these alone cannot recover the key.

The NTT spreads the fault ML-KEM samples $\hat{\mathbf{A}}$ directly in the NTT domain. A small coefficient-domain fault $\Delta\mathbf{A}$ must therefore be injected as $\Delta\hat{\mathbf{A}} = \mathsf{NTT}(\Delta\mathbf{A})$. A single coefficient change in $\Delta\mathbf{A}$ affects every coefficient of $\Delta\hat{\mathbf{A}}$, so the fault needs a very large number of bit flips. Moreover, $\Delta\mathbf{A}$ is multiplied by both $\mathbf{s}$ and $\mathbf{r}$, so the induced error grows quickly with its magnitude. The vector $\mathbf{e}$ has neither problem. It must be sampled from the Centered Binomial Distribution in the coefficient domain, so a sparse fault can be injected directly in Line 5 of Algorithm 1.

4 Experiments and Hardware Evaluation

We evaluate Keyber in two parts. In Section 4.1 we recover the full secret key in simulation, across all three ML-KEM parameter sets and their fault magnitudes. In Section 4.2 we realise the required fault on a running device for ML-KEM-512, and explain why the other parameter sets stay out of reach for clock-based fault injection. In every case the attack

uses a single fault in KeyGen, and the number of inequalities the adversary collects is independent of the number of faults.

4.1 Simulation

All simulations run on a Linux x86_64 workstation with an Intel Core Ultra-7 processor and 32 GB RAM, under Python 3.14. We use NumPy and SciPy for the linear algebra, PyCryptodome for the ML-KEM primitives, and the Roulette solver [Del22] to solve the collected inequalities. We adapt the Roulette reference[1]. which targets Kyber, to ML-KEM as standardised in FIPS 203. The faults we inject (as shown in Section 3.4) for each parameter set, along with the number of inequalities solved for successful key recovery, are shown in Table 2.

Table 2: # Inequalities needed to recover the full key using the chosen fault magnitudes for each parameter set. The extreme value that $e_{ind}[i]$ reaches is $\pm\eta_1 c$ for a single-coefficient fault and $\pm 2\eta_1 c$ for a two-coefficient fault.

Parameter set	**# Faulted Coefficients**	**Fault magnitude c**	**Induced extremum**	**# Inequalities**
ML-KEM-512	1	256	± 768	$\approx$ 19,000
	2	128	± 768	$\approx$ 20,000
ML-KEM-768	2	192	± 768	$\approx$ 25,000
		208	± 832	$\approx$ 20,000
ML-KEM-1024	2	192	± 768	$\approx$ 30,000
		208	± 832	$\approx$ 15,000

Inequality generation After the faulty KeyGen, the adversary runs honest encapsulations against the faulty public key and keeps only the ciphertexts whose induced error e_{ind} reaches its extremum at exactly one coefficient (rejection sampling, Section 3.3). For such an accepted ciphertext, let the extremum fall on the k^{th} coefficient of e_{ind} with value $\pm E$. Combining both decapsulation outcomes of Equation (4), we get

$$|e_{ind}[k] + e_{std}[k]| \lessgtr \frac{q}{4}.$$

We consider the case where the extremum was a maximum and expand as follows:

$$\begin{aligned} e_{ind}[k] + e_{std}[k] &\lessgtr \frac{q}{4} \quad \textbf{or} \quad e_{ind}[k] + e_{std}[k] \gtrless -\frac{q}{4} \\ E + e_{std}[k] &\lessgtr \frac{q}{4} \quad \textbf{or} \quad E + e_{std}[k] \gtrless -\frac{q}{4} \\ e_{std}[k] &\lessgtr \frac{q}{4} - E \quad \textbf{or} \quad e_{std}[k] \gtrless -\frac{q}{4} - E \end{aligned}$$

We know that $|e_{std}[k]| < \frac{q}{4}$. The inequality on the right is therefore redundant and can be ignored. Hence, if the injected fault causes a positive effect on the induced error, we derive the following inequality, depending on the decapsulation outcome.

$$e_{std}[k] \begin{cases} < \frac{q}{4} - E & \text{if Decaps succeeds} \\ > \frac{q}{4} - E & \text{if Decaps fails} \end{cases} \tag{6}$$

[1] https://github.com/Crypto-TII/roulette

Similarly, in the case where the extremum was a minimum, we ignore the redundant inequality on the left and derive the following inequality.

$$e_{std}[k] \begin{cases} > -\frac{q}{4} + E & \text{if Decaps succeeds} \\ < -\frac{q}{4} + E & \text{if Decaps fails} \end{cases} \tag{7}$$

The right-hand sides of Equations (6) and (7) can be simplified by defining $\nu = \frac{q}{4} - E$. Thus, we use ν and $-\nu$ for inequalities generated by an extremum of $+E$ and $-E$, respectively.

We know that the unknowns $\mathbf{s}$ and $\mathbf{e}$ appear linearly in $e_{std}[k]$ through polynomial multiplication. The k-th coefficient of a polynomial-product in R_q can be equivalently represented as an inner product. Precisely, we take the inner product of the k-th row (denoted as $\mathbf{A}^{(k)}$) of the *anti-circulant* matrix of one polynomial with another polynomial vector. Thus, the k-th coefficient of the standard decryption error can be simplified as follows:

$$\begin{aligned} e_{std}[k] &= (\delta v - \mathbf{s}^\top \delta \mathbf{u})[k] + (\mathbf{e}^\top \mathbf{r} + e_2 - \mathbf{s}^\top \mathbf{e_1})[k] \\ &= (\mathbf{s}^\top(-\mathbf{e_1} - \delta \mathbf{u}))[k] + (\mathbf{e}^\top \mathbf{r})[k] + (e_2 + \delta v)[k] \\ &= \left\langle \mathbf{s}, -\mathbf{e_1}^{(k)} - \delta \mathbf{u}^{(k)} \right\rangle + \left\langle \mathbf{e}, \mathbf{r}^{(k)} \right\rangle + (e_2 + \delta v)[k] \\ &= \left\langle -\mathbf{e_1}^{(k)} - \delta \mathbf{u}^{(k)}, \mathbf{s} \right\rangle + \left\langle \mathbf{r}^{(k)}, \mathbf{e} \right\rangle + (e_2 + \delta v)[k] \\ &= \begin{bmatrix} -\mathbf{e_1}^{(k)} - \delta \mathbf{u}^{(k)} & \mathbf{r}^{(k)} \end{bmatrix} \begin{bmatrix} \mathbf{s} \\ \mathbf{e} \end{bmatrix} + (e_2 + \delta v)[k]. \end{aligned}$$

In general, for the i-th decapsulation query, $e_{std}[k]$ is represented as:

$$\begin{bmatrix} (-\mathbf{e_1}^i)^{(k_i)} - (\delta \mathbf{u}^i)^{(k_i)} & (\mathbf{r}^i)^{(k_i)} \end{bmatrix} \begin{bmatrix} \mathbf{s} \\ \mathbf{e} \end{bmatrix} + \left[e_2^i[k_i] + \delta v^i[k_i] \right].$$

Thus, an adversary builds a system of m inequalities of the following form

$$\begin{bmatrix} (-\mathbf{e_1}^1)^{(k_1)} - (\delta \mathbf{u}^1)^{(k_1)} & (\mathbf{r}^1)^{(k_1)} \\ (-\mathbf{e_1}^2)^{(k_2)} - (\delta \mathbf{u}^2)^{(k_2)} & (\mathbf{r}^2)^{(k_2)} \\ \vdots & \vdots \\ (-\mathbf{e_1}^m)^{(k_m)} - (\delta \mathbf{u}^m)^{(k_m)} & (\mathbf{r}^m)^{(k_m)} \end{bmatrix} \begin{bmatrix} \mathbf{s} \\ \mathbf{e} \end{bmatrix} + \begin{bmatrix} e_2^1[k_1] + \delta v^1[k_1] \\ e_2^2[k_2] + \delta v^2[k_2] \\ \vdots \\ e_2^m[k_m] + \delta v^m[k_m] \end{bmatrix} \lesseqgtr \begin{bmatrix} \pm\nu \\ \pm\nu \\ \vdots \\ \pm\nu \end{bmatrix}. \tag{8}$$

In the above system, we call the left matrix A and the unknowns $\begin{bmatrix} \mathbf{s} & \mathbf{e} \end{bmatrix}^\top$ as x. We combine the two constant matrices on either side of the inequality and call it B, thus forming a system of inequalities of the form $Ax + B \lesseqgtr 0$. Since the decapsulation outcome only determines the direction of the inequality, the adversary can precompute A and B before querying the decapsulation oracle, as described in Algorithm 2. Once the decapsulation outcome is known, the inequality is completely determined and fed to the solver.

Secret recovery We pass systems of increasing size to the Roulette solver and record the fraction of secret coefficients recovered. Figure 4 plots the recovery rate against the number of inequalities for each parameter set. Full recovery needs between 15,000 and 30,000 inequalities, depending on the parameter set and fault magnitude, as listed in Table 2. The single-coefficient fault on ML-KEM-512 achieves full key recovery at about 19,000 inequalities.

4.2 Hardware Evaluation

We now realise the ML-KEM-512 single-coefficient fault on a real device.

Algorithm 2 Inequality Generation

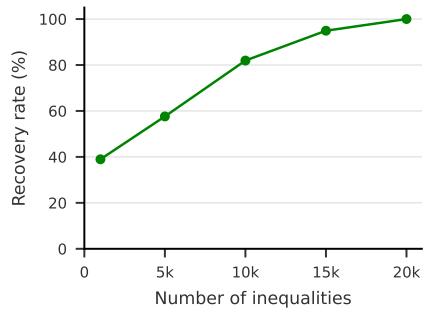
Input: faulty key pk^*, fault $\Delta\mathbf{e}$, number of inequalities m
Output: inequality rows A, constants B, ciphertexts CT, session keys SS, boundary flags BFlag

$E \leftarrow$ extreme value of e_{ind} ▷ $\eta_1 c$ (single coeff.) or $2\eta_1 c$ (two coeff.)
$\nu \leftarrow \frac{q}{4} - E$
$ctr \leftarrow 0$
Initialise empty arrays $A, B, \mathsf{CT}, \mathsf{SS}, \mathsf{BFlag}$
while $ctr < m$ **do**
  $ct, ss, \mathbf{r}, \mathbf{e_1}, e_2, \delta\mathbf{u}, \delta v \leftarrow \mathsf{Kyber.Encaps}(\mathsf{pk}^*)$
  $e_{ind} \leftarrow \Delta\mathbf{e}^\top \mathbf{r}$ ▷ Known induced error
  $cnt \leftarrow \mathsf{FindCount}(e_{ind}, +E) + \mathsf{FindCount}(e_{ind}, -E)$ ▷ No. of extreme values
  if $cnt == 1$ **then**
    $\mathsf{CT}[ctr] \leftarrow ct$
    $\mathsf{SS}[ctr] \leftarrow ss$
    if $\mathsf{FindCount}(e_{ind}, +E) == 1$ **then**
      $k \leftarrow \mathsf{FindIndex}(e_{ind}, +E)$
      $B[ctr] \leftarrow (e_2 + \delta v)[k] - \nu$
      $\mathsf{BFlag}[ctr] \leftarrow 0$ ▷ Boundary at $+\nu$
    else
      $k \leftarrow \mathsf{FindIndex}(e_{ind}, -E)$
      $B[ctr] \leftarrow (e_2 + \delta v)[k] + \nu$
      $\mathsf{BFlag}[ctr] \leftarrow 1$ ▷ Boundary at $-\nu$
    end if
    $A[ctr] \leftarrow \mathsf{AntiCirc}(-\mathbf{e_1} - \delta\mathbf{u})[k] \,\|\, \mathsf{AntiCirc}(\mathbf{r})[k]$
    $ctr \leftarrow ctr + 1$
  end if
end while
return $A, B, \mathsf{CT}, \mathsf{SS}, \mathsf{BFlag}$

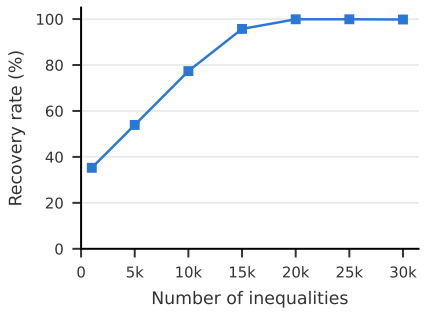
Platform and profiling The fault is produced on a ChipWhisperer-Husky capture device driving a CW308 UFO baseboard with a CW308_STM32F3 target (STM32F303, ARM Cortex-M4F). The target runs a reference FIPS 203 ML-KEM-512 implementation[2] in which KeyGen is called on fixed coins, and a trigger brackets the CBD sampling and store of the error polynomial $\mathbf{e_0}$. The stored coefficients are read back and checked bit-exact against the reference, so every faulted coefficient is fully attributed. The core is clocked from the capture device at a nominal 20 MHz. Sweeping the clock upward, the target is clean up to $\approx$ 40 MHz, faults the stored coefficients of $\mathbf{e_0}$ in a narrow band around 40.0–40.6 MHz, and hangs above $\approx$ 42 MHz. The timing violation injects an additive +256 (the bit-8 weight 2^8) into a coefficient during CBD output. The perturbation is a short burst: one coefficient takes the full +256 carrier and a few neighbours take small $\pm 1 \ldots 4$ deviations. We keep the shots in which exactly one coefficient carries the +256, the single-carrier condition the attack needs. Our working instance (captured at 40.25 MHz) changes 23 coefficients of $\mathbf{e_0}$: the carrier $e_0[227] : -1 \to +255$ (an additive +256, so $e_{ind} = 3 \times 256 = 768$) and 22 junk coefficients with $|\Delta| \le 4$, all inside the window $[227, 255]$.

Reproducibility The fault is deterministic for a fixed (key, frequency): rerunning with the same KeyGen coins reproduces it bit-for-bit. Sweeping the frequency walks the carrier across different coefficients, in both signs, and reflashing with different coins moves it to a new coefficient of a new key. Table 3 lists the single-carrier faults captured this way, across

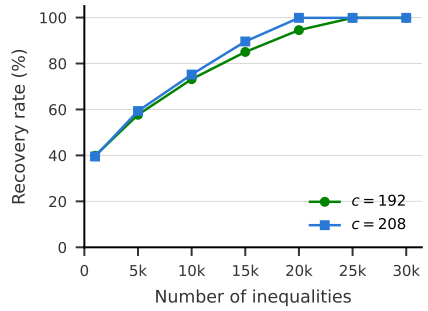
[2] https://github.com/pqclean/PQClean



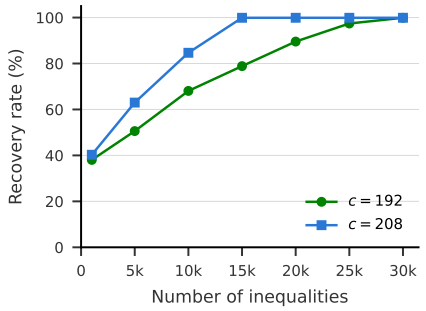
(a) ML-KEM-512, single coefficient ($c = 256$)



(b) ML-KEM-512, two coefficients ($c = 128$)



(c) ML-KEM-768, two coefficients ($c = 192, 208$)



(d) ML-KEM-1024, two coefficients ($c = 192, 208$)

Figure 4: Coefficient recovery rate against the number of inequalities, from a single fault in KeyGen. Each curve is one fault magnitude c. Full recovery is reached between 15,000 and 30,000 inequalities.

two keys and five carrier positions. The carrier is always an additive ± 256; the number of co-perturbed junk coefficients $|S|$ varies with the operating point.

End-to-end recovery We verify recovery end to end. The fault is physical; the decapsulation-failure oracle and the solver are software, driven by the hardware-measured fault pattern. For each captured fault the full measured pattern (carrier and all junk) is injected into the software key, the adversary models only the carrier, and the software Decaps provides the oracle. The generated inequalities agree with the reference Decaps on every row. Table 4 reports the recovered fraction of $\mathbf{s}$: every captured fault recovers the complete ML-KEM-512 secret, independent of key, carrier position, sign, or burst size $|S|$. The junk is harmless, as it corrupts only low bits, so the non-carrier coefficients stay within $[-\eta_1, \eta_1]$ and the solver recovers the true $\mathbf{e_0}$. This closes the loop: a single over-clock fault in KeyGen yields full ML-KEM-512 key recovery.

Other parameter sets For ML-KEM-768 and ML-KEM-1024 ($\eta_1 = 2$) the attack needs a two-coefficient fault whose induced error reaches close to $\frac{q}{4}$, which calls for precise additive magnitudes such as $+128$ per coefficient. Clock-based injection does not deliver these. On a Cortex-M4 target a clock glitch reaches only quantised magnitudes that skip the required value, and on an ML-KEM-768 FPGA core an over-clock flips single bits that add 512, 640, or 1024 to a coefficient, missing the $\frac{q}{4}$ window. We confirmed by register-level emulation that the attack itself succeeds once the exact magnitude is supplied, recovering 1534 of 1536 coefficients at $c = 208$, so the only gap is the injection method and not the attack.

Table 3: Single-carrier over-clock faults captured on the CW308_STM32F3, across two KeyGen keys and several frequencies. The carrier is always an additive ±256; the burst size $|S|$ (coefficients changed, including the carrier) varies with the operating point.

Key (KeyGen coins)	Over-clock	Carrier	±	$\lvert S\rvert$
fixed	40.05 MHz	$e_0[111]$	+256	113
fixed	40.25 MHz	$e_0[227]$	+256	23
fixed	39.50 MHz	$e_0[97]$	−256	122
fixed	40.40 MHz	$e_0[10]$	+256	190
fixed	40.40 MHz	$e_0[1]$	−256	196
second key	39.90 MHz	$e_0[109]$	−256	106

Table 4: Secret-key recovery ($\mathbf{s}$ coefficients, out of 512) from the real over-clock faults, using the Roulette solver on a software Decaps oracle of the faulty key. Full recovery is reached at 19,000 inequalities for the swept faults, independent of key, carrier position, sign, or burst size $|S|$. The carrier $e_0[227]$ is our working instance, validated to full recovery at 20,000 inequalities.

Key	Carrier	$\lvert S\rvert$	15,000	19,000	22,000
fixed	$e_0[227] = +256$	23	–	–	512/512
fixed	$e_0[97] = -256$	122	488/512	512/512	512/512
fixed	$e_0[10] = +256$	190	499/512	512/512	512/512
fixed	$e_0[1] = -256$	196	502/512	512/512	512/512
second key	$e_0[109] = -256$	106	505/512	512/512	512/512

Reaching the exact multi-bit pattern on these parameter sets likely requires a spatially and temporally precise technique such as laser fault injection or electromagnetic fault injection. We leave a fully physical attack on ML-KEM-768 and ML-KEM-1024 as an open problem.

5 Countermeasures

We discuss two countermeasures, in increasing order of strength.

Range check The simplest countermeasure is a range check during key generation. After sampling $\mathbf{e}$, the device checks that every coefficient lies in $[-\eta_1, \eta_1]$ and aborts otherwise. A valid CBD sample always passes, and the fault enlarges a coefficient well beyond this range, so the check catches it. The check is cheap, but it guards only the sampling step. A fault injected after the check, or on the stored $\mathbf{e}$ during the NTT, is not caught.

Pairwise consistency test A stronger countermeasure is a pairwise consistency test at the end of key generation. The device encapsulates with the new public key and decapsulates with the new private key, then checks that the two shared secrets agree. A faulty public key makes a random honest ciphertext decapsulate incorrectly with some probability p, so a single test round detects the fault with probability p, and running the test r times raises the detection probability to $1 - (1 - p)^r$.

This exposes a trade-off for the adversary. A larger fault gives a cleaner balance of successes and failures, and therefore faster key recovery, but it is also easier for a consistency test to detect. The ML-KEM-512 faults are the hardest to detect, which comes at the cost of a lower yield.

6 Conclusion

Through the presented work, we answered the question positively: whether a faulty public key can be exploited using statistical methods without requiring any additional faults. To produce a faulty public key, we explored two possible fault locations in key generation of ML-KEM, viz. the polynomial matrix and the error polynomials. The latter led to a successful fault attack. Our attack emphasised that the controlled propagation of a fault can lead to a meaningful shift ($\pm q/4$) in any coefficient of the noisy message after decryption. As the attack requires only a single fault, it differs from existing Statistical Ineffective Fault Attacks, which require thousands of fault injections during decapsulation.

We validated the attack in simulation across all parameter sets, achieving full-key recovery with up to 30,000 inequalities using the Roulette solver. For ML-KEM-512, we also realise the required fault on hardware through an over-clock and recover the entire secret key end-to-end. The larger parameter sets require a more precise multi-bit fault that clock-based injection cannot achieve. We leave a fully physical attack on these sets as an open problem and believe it can be demonstrated using a laser or an electromagnetic fault-injection setup.

AI Usage Disclosure

The authors conceived the attack, performed the simulations validating its correctness, obtained the reported results, and prepared the manuscript. Claude Opus (Anthropic) contributed substantially to the hardware evaluation of the attack. It also assisted with minor restructuring of the manuscript and carried out a full verification of its contents. All AI-assisted outputs were reviewed by the authors, who take full responsibility for the work.

Acknowledgments. We thank Anirudh Jaiswal for a basic fault-injection setup that was crucial to completing the hardware evaluations.

References

[BDK+18] Joppe W. 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*, pages 353–367. IEEE Computer Society Press, April 2018.

[BDL97] Dan Boneh, Richard A. DeMillo, and Richard J. Lipton. On the importance of checking cryptographic protocols for faults (extended abstract). In Walter Fumy, editor, *EUROCRYPT'97*, volume 1233 of *LNCS*, pages 37–51. Springer, Berlin, Heidelberg, May 1997.

[BS97] Eli Biham and Adi Shamir. Differential fault analysis of secret key cryptosystems. In Burton S. Kaliski, Jr., editor, *CRYPTO'97*, volume 1294 of *LNCS*, pages 513–525. Springer, Berlin, Heidelberg, August 1997.

[Del22] Jeroen Delvaux. Roulette: A diverse family of feasible fault attacks on masked Kyber. *IACR TCHES*, 2022(4):637–660, 2022.

[FO99] Eiichiro Fujisaki and Tatsuaki Okamoto. How to enhance the security of public-key encryption at minimum cost. In Hideki Imai and Yuliang Zheng, editors,

PKC'99, volume 1560 of *LNCS*, pages 53–68. Springer, Berlin, Heidelberg, March 1999.

[HMS+23] Julius Hermelink, Erik Mårtensson, Simona Samardjiska, Peter Pessl, and Gabi Dreo Rodosek. Belief propagation meets lattice reduction: Security estimates for error-tolerant key recovery from decryption errors. *IACR TCHES*, 2023(4):287–317, 2023.

[HPP21] Julius Hermelink, Peter Pessl, and Thomas Pöppelmann. Fault-enabled chosen-ciphertext attacks on Kyber. In Avishek Adhikari, Ralf Küsters, and Bart Preneel, editors, *INDOCRYPT 2021*, volume 13143 of *LNCS*, pages 311–334. Springer, Cham, December 2021.

[KCS+24] Suparna Kundu, Siddhartha Chowdhury, Sayandeep Saha, Angshuman Karmakar, Debdeep Mukhopadhyay, and Ingrid Verbauwhede. Carry your fault: A fault propagation attack on side-channel protected LWE-based KEM. *IACR TCHES*, 2024(2):844–869, 2024.

[Nat24] National Institute of Technology and Standards. Module-Lattice-Based Key-Encapsulation Mechanism Standard. Federal Information Processing Standards Publication (FIPS) NIST FIPS 203, U.S. Department of Commerce, Washington, D.C., August 2024.

[PP21] Peter Pessl and Lukas Prokop. Fault attacks on CCA-secure lattice KEMs. *IACR TCHES*, 2021(2):37–60, 2021.

[RCDB24] Prasanna Ravi, Anupam Chattopadhyay, Jan Pieter D'Anvers, and Anubhab Baksi. Side-channel and fault-injection attacks over lattice-based post-quantum schemes (kyber, dilithium): Survey and new results. *ACM Trans. Embed. Comput. Syst.*, 23(2), March 2024.

[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 Ilia Polian and Marc Stöttinger, editors, *COSADE 2019*, volume 11421 of *LNCS*, pages 232–250. Springer, Cham, April 2019.

[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 TCHES*, 2023(2):447–481, 2023.

[WCCL25] Jian Wang, Weiqiong Cao, Hua Chen, and Haoyuan Li. Secret in OnePiece: Single-bit fault attack on kyber. In Christina Boura, Atefeh Mashatan, and Ali Miri, editors, *SAC 2025*, volume 16207 of *LNCS*, pages 781–804. Springer, Cham, August 2025.