Deterministic Secret Seed : Practical Fault Injection on Kyber's Implementations

Guillaume Blanc[1,2], Abdellah Bouagoun[2], Mosabbah Mushir Ahmed[2], and Alieeldin Mady[2]

[1] EURECOM, Sophia-Antipolis, France
guillaume.blanc@eurecom.fr
[2] Qualcomm Technologies, Cork, Ireland
{abouagou,mosabbah,amady}@qti.qualcomm.com

Abstract. CRYSTALS-Kyber has been an interesting target for many hardware attacks and more so after its standardization as ML-KEM. In this paper, we cover a practical fault attack against its implementations. We demonstrate that a simple fault during the generation of the seeds can result in a seed reuse scenario, which can be exploited to recover the secret key from multiple key generations. We performed experiments on ARM and RISC-V architectures, against the reference implementation, *pqm4*, an ARM Cortex-M4 optimized implementation, and *mlkem-native*, a recently released, secure and widely used implementation. Although *mlkem-native* is expected to see broad deployment, there is currently no research about faulting it in literature. We used voltage glitching to skip a single instruction, leading to the secret recovery. This proves the importance of securing the generation of the seeds, being at the core of cryptographic operations, at the implementation level. Even though this attack is implementation dependent, to our knowledge, no implementation is secured against it.

Keywords: Kyber · Fault injection · Voltage glitching

1 Introduction

1.1 Overview

With the rise of quantum computers, the threat of having such a large computer, capable of breaking modern asymmetric cryptography has become increasingly unavoidable. Research on post-quantum cryptographic algorithms was driven by the National Institute of Standards and Technology (NIST) with the launch of the Post-Quantum Cryptography (PQC) Standardization project [5]. This competition aimed to establish new quantum-resistant key-encapsulations mechanisms (KEMs) and signature schemes. Kyber, a lattice-based KEM was one of the candidates, and was standardized in FIPS 203 under the name of Module-Lattice-Based Key-Encapsulation Mechanism (ML-KEM) [11].

Several implementations of Kyber are available in the literature. The reference implementation [2] from PQ Crystals, the authors of Kyber, is one of them.

Its security is crucial as many other implementations are forking it, like *pqm4* [4], an ARM Cortex-M4 optimized implementation. There are also secure implementations, such as *mlkem-native* [7], which was made by PQ Code Package, a project from the PQC Alliance, as part of the Linux Foundation. Even if the scheme is chosen-ciphertext attack secure, and the implementation memory-safe, hardware attacks are still a possible vector.

In the field of hardware attacks, attackers exploit physical vulnerabilities to bypass protections or to extract sensitive information. One such method is Fault Injection (FI), which consists of deliberately inducing a fault in a system to alter its behavior. Among the various FI techniques, Voltage Glitching (VG) is particularly notable for its simplicity. It works by altering the power supply voltage of a device during its operation, allowing an attacker to skip instructions, bypass security checks or corrupt computations.

1.2 Our Contribution

In this work, we try to mount another attack targeting the key generation, to illustrate how fault attacks can produce weak instances of Kyber that can be broken. We show that a single fault injected on multiple key generations allows an attacker to recover the secret from encapsulation keys. We also prove the practicality of this attack on three implementations and perform experimental validation on two architectures.

The contribution of this work is as follows:

- We present a secret recovery by performing fault injection to skip a single instruction from Kyber's key generation.
- We perform an experimental validation by voltage glitching ARM and RISC-V devices with the reference implementation [2], *pqm4* [4] and *mlkem-native* [7].

1.3 Related Work

Various FI attacks against Kyber's implementations already exist, enabling the recovery of secrets by faulting the key generation [10], the decapsulation [8,3], or by reducing the scheme's security to chosen-plaintext attacks [12]. Ravi et al. [9] present a fault attack targeting both key generation and encapsulation. They exploit the reuse of a seed to sample different elements with a small domain separator only. By injecting a fault to make the domain separator reused, the resulting instance becomes trivially solvable. Although this attack was published before the standardization of Kyber, it remains reproducible. In this work, rather than faulting the domain separator, we target the generation of the seeds themselves, leading to a seed-reuse scenario. We exploit it by having two instances generated with the same seed, which can again be trivially solved. While they performed their experiments using Electromagnetic Fault Injection (EMFI), we used VG to inject a fault. A summarized comparison is available in Table 1.

Table 1. Comparison between Number "Not Used" Once and Deterministic Secret Seed

	Number "Not Used" Once [9]	**Deterministic Secret Seed**
Targeted part	Key generation, Encapsulation	Key generation
Fault objective	Nonce reuse	Seed reuse
Number of faults	4–8	2
Fault injection technique	EMFI	VG
Target architecture	ARM	ARM, RISC-V
Target implementation	*pqm4*	Reference, *pqm4*, *mlkem-native*
Recovery success rate	100%	15–96%

1.4 Outline

In Section 2, we introduce Kyber and the mathematical objects in use. In Section 3, we explain the mathematical background of the attack, the weaknesses in the implementations that allows to perform the attack and the exploitation. Then, in Section 4, we prove the practicality of the attack by experimenting it under different conditions.

2 Preliminaries

We denote the polynomial ring $\mathbb{Z}_q[X]/(X^n+1)$ as R_q where $n, q \in \mathbb{N}$. Arrays of p bytes are denoted $\mathcal{B}^p$ and the concatenation is $\|$. For $m, n \in \mathbb{N}$, the set of vectors of m polynomials is denoted R_q^m and the set of $m \times n$ matrices is denoted $R_q^{m \times n}$. The Number Theoretic Transform (NTT) of a polynomial or a vector is denoted with a hat (e.g., $\forall p \in R_q, \widehat{p} = \mathrm{NTT}(p)$). The operation $\cdot$ denotes standard matrix multiplication.

Kyber is based on the hardness of the Module-Learning-With-Error (MLWE) problem [1]. It comes with three parameter sets: ML-KEM-512, ML-KEM-768 and ML-KEM-1024. A key parameter affected by these sets is $k \in \{2, 3, 4\}$, which represents the size of elements involved in Kyber's operations. The public elements are $A \in R_q^{k \times k}$ and $t \in R_q^k$, while $s, e \in R_q^k$ remain private. These elements are related by the equation:

$$t = A \cdot s + e \tag{1}$$

During the key generation, $\rho \in \mathcal{B}^{32}$ serves as the seed to sample A, we denote this operation Expand. The secret s and the error e are sampled from $\sigma \in \mathcal{B}^{32}$, with a domain separator to ensure they are not equal, we denote this operation SamplePolyCBD. These operations are represented in the following equations, using $N, N' \in \mathbb{N}, N \neq N'$ as nonces:

$$\begin{aligned} &\rho \in \mathcal{B}^{32}, \quad A = \mathrm{Expand}(\rho) \\ &\sigma \in \mathcal{B}^{32}, \quad s = \mathrm{SamplePolyCBD}(\sigma, N), \quad e = \mathrm{SamplePolyCBD}(\sigma, N') \end{aligned} \tag{2}$$

3 Fault Attack

The core idea of our attack is to create a seed reuse scenario. FI is used to generate two key pairs having private components in common. During two key generations, the seed ρ must differ while the seed σ must be the same. It implies different public elements A and t, while the private ones s and e are reused. Under this condition, the instances of Kyber become trivially solvable, and the secret can be recovered from the encapsulation key.

3.1 Background

We assume two instances of Kyber, with the seeds $\rho_0, \rho_1, \sigma_0, \sigma_1$, as there are two unknowns, and we require to eliminate the error e. From Equation 1, we have:

$$\begin{cases} t_0 = A_0 \cdot s_0 + e_0 \\ t_1 = A_1 \cdot s_1 + e_1 \end{cases}$$

Then, a fault is injected such that $\sigma_0 = \sigma_1$. From Equation 2, it implies that the secrets and the errors are reused: $s = s_0 = s_1$ and $e = e_0 = e_1$. As a result, the instances can be substituted:

$$t_1 - t_0 = (A_1 - A_0) \cdot s$$

Let us define $\alpha = A_1 - A_0 \in R_q^{k \times k}$ and $\theta = t_1 - t_0 \in R_q^k$. If α is invertible, we can solve the equation for s:

$$s = \alpha^{-1} \cdot \theta$$

Otherwise, if α is not invertible, the current instances are dropped, and new ones are generated. There is a high chance of α being invertible because it is the difference of two random and independent matrices over a sufficiently large coefficient domain. While it would still be possible to resolve this case using a pseudo-inverse, we opt to generate new instances for simplicity. To invert the matrix, we used the Extended Euclidean Algorithm for inversion of polynomials in R_q.

In general, we can set that given two key generations indexed $i, j \in \mathbb{N}$, with the private seeds σ_i, σ_j and the public seeds ρ_i, ρ_j. Let us define $\alpha_{i,j} = A_i - A_j$ and $\theta_{i,j} = t_i - t_j$. If $\sigma_i = \sigma_j$ and $\rho_i \neq \rho_j$ and $\alpha_{i,j}$ is invertible, then

$$s_i = s_j = \alpha_{i,j}^{-1} \cdot \theta_{i,j} \tag{3}$$

3.2 Seeds in Implementations

In this section, we explain then how the seeds are generated and how to maintain a constant σ value among two key generations. The seeds ρ and σ are generated by hashing random bytes $d \in \mathcal{B}^{32}$ with k appended as a domain separator, using the G function (i.e., SHA3-512) [11].

```
void mlk_keccakf1600_extract_bytes(uint64_t *state, unsigned
    char *data, unsigned offset, unsigned length)
{
  unsigned i;
  uint8_t *state_ptr = (uint8_t *)state + offset;
  for (i = 0; i < length; i++)
  __loop__(invariant(i <= length))
  {
    data[i] = state_ptr[i];
  }
}
```

Listing 1. mlk_keccakf1600_extract_bytes function

Skip the Hash In both the reference implementation [2] and *pqm4* [4], the input and the output of the hash function are stored in the same 64-bytes array, `buf`. This input is made of 33 bytes, which lets 31 uninitialized bytes in `buf`, denoted as $x \in \mathcal{B}^{31}$. Before the hash, `buf` contains then $d\|k\|x$. Once hashed, the 32 first bytes of `buf` are used as ρ while the 32 last bytes are used as σ.

We propose injecting a fault to skip the hash, resulting in `buf` not being altered by the hash function. Instead of using the hash as the seeds, the input of G is used, i.e. $d\|k\|x$. Then, with the fault, $\rho = d$ and $\sigma = k\|x$, in other words, ρ is random while σ is not. The attack relies on the redundancy of x across multiple generations to compute Equation 3.

Break the Loop An alternative method to achieve the same outcome is to skip the writing of `buf` within the hash function, instead of skipping the hash. This writing is done in a `for` loop. In *mlkem-native* [7], this loop is included in the `mlk_keccakf1600_extract_bytes` function, provided in Listing 1. In this implementation, the hash is computed in `state`, then it is copied into `data`, which represents `buf`. We propose to inject a fault to break this loop before its 33^{rd} iteration, in order to write ρ without writing σ. However, it is crucial that the loop is not broken before writing at least one byte, otherwise, there is a risk that ρ may also become constant, and the conditions from Equation 3 would not be satisfied. While this would prevent secret recovery via our specific method, it would instead result in a key reuse scenario, potentially exposing the system to alternative attacks.

In *mlkem-native* [7], skipping the hash is not applicable because separate arrays are used for the input and output of the hash function; doing so would leave the output buffer uninitialized. While these values might be predictable in some environments, they prevent ρ from remaining constant across different instances, which is a requirement for solving Equation 3. Instead, it is necessary to break the loop, to ensure a new ρ is generated while σ remains filled with uninitialized values.

3.3 Secret Recovery after Fault

We propose Algorithm 1 to recover a secret shared from two faulted encapsulation keys and to verify the validity of the recovered secret. This algorithm makes no prior assumptions, meaning that it can handle cases where the conditions from Equation 3 are not satisfied. It takes as input the encapsulation keys of two different key generations, denoted by indices i and j. We used this algorithm in Section 4 with different implementations and architectures to find the manner of selecting these iterations. For now, we consider a key generation device that is continuously running and faulted for each generation. The last generation is indexed by i, which corresponds to the secret we want to recover. The algorithm is used in a brute-force manner, using all previously generated encapsulation keys (e.g., $j \in [0, i)$), to try recovering the current secret. The function H denotes SHA3-256, as from the specification.

Algorithm 1 Secret Recovery from Faulted Encapsulation Keys

Require: $ek_i = (\widehat{t_i}, \rho_i), ek_j = (\widehat{t_j}, \rho_j)$ for $i, j \in \mathbb{N}$
$t_i \leftarrow \mathrm{NTT}^{-1}(\widehat{t_i})$
$t_j \leftarrow \mathrm{NTT}^{-1}(\widehat{t_j})$
$A_i \leftarrow \mathrm{Expand}(\rho_i)$
$A_j \leftarrow \mathrm{Expand}(\rho_j)$
$\alpha_{i,j} \leftarrow A_j - A_i$
$\theta_{i,j} \leftarrow t_j - t_i$
if $\alpha_{i,j}$ is invertible **then**
    $s^* \leftarrow \alpha_{i,j}^{-1} \cdot \theta_{i,j}$
    $K, c \leftarrow \mathrm{Encaps}(ek_i)$
    $dk^* \leftarrow (s^*, ek_i, \mathrm{H}(ek_i), \{0\}^{32})$
    $K' \leftarrow \mathrm{Decaps}(dk^*, c)$
    if $K = K'$ **then**
        return s^*
    end if
end if
return Recovery Failed

4 Experimental Validation

4.1 Setup

In this section, we perform experiments of our attack to verify its feasibility and to evaluate the redundancy of uninitialized values. Since the outcome of the attack is implementation and device dependent, three scenarios are evaluated.

To perform the VG, we use a ChipWhisperer Lite[6] (CW) as the glitcher. The first target is an STM32F415 microcontroller, running at 24 MHz, mounted on a CW308T board, and denoted STM. We selected this device for its ARM

architecture, allowing to run *pqm4* [4]. We also ran *mlkem-native* [7] on the STM to compare the results across implementations. We ported both implementations to the STM, using the SimpleSerial protocol[3] from ChipWhisperer for communication.

The second target is an ESP32-C3-MINI-1 mounted on a Beetle ESP32-C3, running at 168 MHz, and denoted ESP. We chose this device for its RISC-V architecture. We altered the board by removing the capacitors from the power rail and by replacing the power supply of the chip with the one of the CW, enhancing its vulnerability to VG. We ported the reference implementation [2] to it, which has the same manner of generating the seeds than *pqm4* [4] codewise. Although the exact same implementation was not used, it allowed us to compare the two architectures.

Each glitch is triggered by a GPIO pin that is toggled before the beginning of the key generation. A glitch is considered successful if the value of σ is not changed by the hash function. Either because the hash was skipped (for the reference implementation [2] and *pqm4* [4]), or the loop writing the output was stopped before writing σ (for *mlkem-native* [7]). It is important to note that a successful glitch does not necessarily lead to successful secret recovery.

4.2 Outcome

We then ran the experiments thousand times. Table 2 summarizes the glitch parameters and the results. For each scenario, we observed different outcomes which are explained in this section. Figure 1 shows oscilloscope traces of the glitches used to fault the STM and ESP.

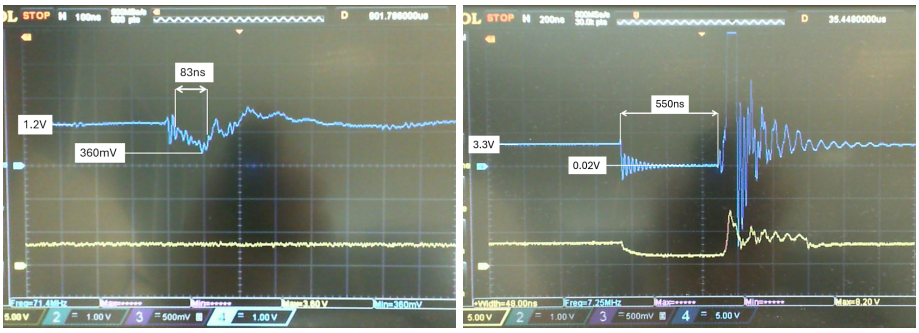

Fig. 1. Glitch shapes to fault the STM (left) and ESP (right)

[3] Documentation available at `https://chipwhisperer.readthedocs.io/en/latest/simpleserial.html`

Scenario 1: STM with pqm4. For the first scenario, we noticed that the consecutiveness of two successful glitches produces the desired conditions (same σ), enabling us to recover the secret. We then used Algorithm 1 to recover the secret using two consecutive encapsulation keys. For the generation $i \in \mathbb{N}$, the secret was computed with $j = i - 1$. Across all attempts, we successfully recovered the current secret about 21% of the time.

Scenario 2: ESP with reference implementation. On the ESP, even if the implementation was similar to *pqm4* [4], we did not notice any direct impact of consecutive successful glitches. However, we observed redundancy across all iterations. Then, to recover the secret, we had to test all previously saved encapsulation keys. We repeatedly ran and glitched the key generation, saving every time the encapsulation key. Then, for each new key (indexed i), we used the recovery algorithm $\forall j \in [0, i)$, until the secret was found. We managed to recover 15% of the secrets overall. We also noticed that the median number of computations needed to recover the secret converged at around 600, which means that we could save only the 600 last encapsulation keys and still recover more than 50% of the secrets. It significantly reduces the number of computations while recovering a considerable number of secrets compared to a full brute-force.

Scenario 3: STM with mlkem-native. This time, we injected faults to break the loop at index 32. In this case, when the fault was successful, σ was constant. Every time the loop was successfully faulted, the value of σ was always `"03 65 01 02 eb 00 00 .. 00"` (27 zero bytes). It is relevant to observe many zeros as this implementation zeroizes buffers after use. With this particular setup, it is not necessary to fault multiple key generations. Every time the fault is successful, the secret is deterministically sampled and thus known. Consequently, a single glitch is sufficient to consistently reproduce the same secret.

4.3 Success rate

As summarized in Table 2, the success rate of the glitch was about one third to skip the hash and 96% to break the loop. In the work of Ravi et al.[9], their success rate was about 100%. Our lower success rate is explained by the difference of fault injection technique. While they used EMFI, we used VG. Compared to EMFI, VG is easier to set up but less effective.

It is important to notice that *mlkem-native* [7], the secure implementation, had the best efficiency. The higher success rate compared to the other scenarios could be explained by the fact that it might be easier to break a loop rather than skipping a function call. Moreover, apart from the glitch success rate being greater, it is also easier to exploit a faulted instance because the secret becomes deterministic. There is no need to solve a lot of equations as in scenario 2 or to have consecutive successful glitches as in scenario 1.

Table 2. Comparison of fault injection results with different setups

Scenario	**1**	**2**	**3**
Implementation	*pqm4* [4]	Reference [2]	*mlkem-native* [7]
Device (architecture)	STM (ARM)	ESP (RISC-V)	STM (ARM)
Fault effect	Skip the hash	Skip the hash	Break the loop
Offset from trigger	59µs	25µs	901µs
Glitch width	83ns	550ns	83ns
Resets	52%	0.8%	9%
Success	32%	38%	96%
Secret recovery	21%	15%	96%

5 Conclusion

In this work, we present practical fault attacks on different implementations of Kyber. We exploit the reuse of the seed of both the secret and the error thanks to the redundancy of uninitialized memory, enabling us to recover the secret from two encapsulation keys. We demonstrate the practicality of the attack through experiments on *pqm4* [4], the reference implementation [2] and *mlkem-native* [7] on ARM and RISC-V architectures.

It is important to note that our experiments were conducted under ideal conditions because the targets were only running Kyber's key generation and communication with the computer. It drastically reduces the modifications made to the memory between each iteration, which does not reflect real-word scenarios where other processes could alter memory.

Compared to the study proposed by Ravi et al.[9], we present an attack requiring only one glitch per key generation, but multiple generations (at least two) are required to recover the secret. However, the attack was less efficient during our experiments, mainly due to the use of VG.

Although *mlkem-native* [7] aims to be used in critical environment, this is the first study exploring its vulnerability to fault injection. Despite its secure design, we demonstrated that it is vulnerable to our attack. This work proves the importance of securing the seed generation of cryptographic operations, even if the implementation is memory safe.

For future work, EMFI could be used to improve the efficiency and reduce the number of resets, countermeasures could be established, implemented and their efficiency evaluated, and more rigorous comparisons across scenarios could be done, such as comparing ARM and RISC-V together or faulting different parts of the same implementation.

References

1. Bos, J.W., Ducas, L., Kiltz, E., Lepoint, T., Lyubashevsky, V., Schanck, J.M., Schwabe, P., Seiler, G., Stehlé, D.: CRYSTALS - kyber: A cca-secure module-lattice-based KEM. In: 2018 IEEE European Symposium on Security and Privacy, EuroS&P 2018, London, United Kingdom, April 24-26, 2018. pp. 353–367.

IEEE (2018). https://doi.org/10.1109/EUROSP.2018.00032, https://doi.org/10.1109/EuroSP.2018.00032
2. pq crystals: Kyber. https://github.com/pq-crystals/kyber, accessed: 2025-04-11
3. Hermelink, J., Pessl, P., Pöppelmann, T.: Fault-enabled chosen-ciphertext attacks on kyber. In: Adhikari, A., Küsters, R., Preneel, B. (eds.) Progress in Cryptology - INDOCRYPT 2021 - 22nd International Conference on Cryptology in India, Jaipur, India, December 12-15, 2021, Proceedings. Lecture Notes in Computer Science, vol. 13143, pp. 311–334. Springer (2021). https://doi.org/10.1007/978-3-030-92518-5_15, https://doi.org/10.1007/978-3-030-92518-5_15
4. Kannwischer, M.J., Rijneveld, J., Schwabe, P., Stoffelen, K.: Pqm4: Post-quantum crypto library for the arm cortex-m4. https://github.com/mupq/pqm4, accessed: 2025-02-06
5. National Institute of Standards and Technology: Post-quantum cryptography standardization. https://csrc.nist.gov/projects/post-quantum-cryptography (2017), accessed: 2025-06-20
6. O'Flynn, C., Chen, Z.D.: Chipwhisperer: An open-source platform for hardware embedded security research. In: Prouff, E. (ed.) Constructive Side-Channel Analysis and Secure Design. pp. 243–260. Springer International Publishing, Cham (2014)
7. Package, P.C.: mlkem-native. https://github.com/pq-code-package/mlkem-native, accessed: 2025-06-12
8. Pessl, P., Prokop, L.: Fault attacks on cca-secure lattice kems. IACR Trans. Cryptogr. Hardw. Embed. Syst. **2021**(2), 37–60 (2021). https://doi.org/10.46586/TCHES.V2021.I2.37-60, https://doi.org/10.46586/tches.v2021.i2.37-60
9. Ravi, P., Roy, D.B., Bhasin, S., Chattopadhyay, A., Mukhopadhyay, D.: Number "not used" once - practical fault attack on pqm4 implementations of NIST candidates. In: Polian, I., Stöttinger, M. (eds.) Constructive Side-Channel Analysis and Secure Design - 10th International Workshop, COSADE 2019, Darmstadt, Germany, April 3-5, 2019, Proceedings. Lecture Notes in Computer Science, vol. 11421, pp. 232–250. Springer (2019). https://doi.org/10.1007/978-3-030-16350-1_13, https://doi.org/10.1007/978-3-030-16350-1_13
10. Ravi, P., Yang, B., Bhasin, S., Zhang, F., Chattopadhyay, A.: Fiddling the twiddle constants - fault injection analysis of the number theoretic transform. IACR Trans. Cryptogr. Hardw. Embed. Syst. **2023**(2), 447–481 (2023). https://doi.org/10.46586/TCHES.V2023.I2.447-481, https://doi.org/10.46586/tches.v2023.i2.447-481
11. of Standards, N.I., Technology: Module-lattice-based key-encapsulation mechanism standard. Federal Information Processing Standards Publication FIPS 203, U.S. Department of Commerce, Washington, D.C. (2024). https://doi.org/10.6028/NIST.FIPS.203, https://doi.org/10.6028/NIST.FIPS.203
12. Xagawa, K., Ito, A., Ueno, R., Takahashi, J., Homma, N.: Fault-injection attacks against nist's post-quantum cryptography round 3 KEM candidates. In: Tibouchi, M., Wang, H. (eds.) Advances in Cryptology - ASIACRYPT 2021 - 27th International Conference on the Theory and Application of Cryptology and Information Security, Singapore, December 6-10, 2021, Proceedings, Part II. Lecture Notes in Computer Science, vol. 13091, pp. 33–61. Springer (2021). https://doi.org/10.1007/978-3-030-92075-3_2, https://doi.org/10.1007/978-3-030-92075-3_2