Secret in OnePiece: Single-Bit Fault Attack on Kyber

Jian Wang[1,2], Weiqiong Cao[1,3], Hua Chen[1], and Haoyuan Li[3]

[1] Trusted Computing and Information Assurance Laboratory, Institute of Software, Chinese Academy of Sciences, Beijing 100190, China
`{wangjian2019, chenhua, caoweiqiong}@iscas.ac.cn`
[2] University of Chinese Academy of Sciences, Beijing 100049, China
[3] Zhongguancun Laboratory, Beijing, China
`lihy@zgclab.edu.cn`

Abstract. The post-quantum key encapsulation mechanism, CRYSTALS-Kyber, has recently been selected by the National Institution of Standards and Technology for standardization, making the study of its implementation security a critical concern. As a widely used countermeasure against implementation attacks, masking has been proven effective in protecting Kyber implementations from side-channel attacks. However, the masking process complicates the original scheme and introduces addtional operations, raising the question of whether these changes open up new attack vectors.

In this paper, we propose a novel fault attack targeting the masked implementation of Kyber, focusing on the masked message decoder. Our generic single-bit fault attack is amplified by masked implementations of Kyber. Moreover, the randomness introduced by masking allows a stuck-at fault to behave as a bit-flip fault with a certain flip probability, thereby simplifying the complexity associated with fault injection. Based on a practical attacker model, we demonstrate that each faulted decapsulation reveals partial information about the secret key in the form of an inequality. By employing a solver based on statistical theory and two filtering techniques, the entire secret key can be efficiently recovered by solving the resulting system of inequalities. Only 36,000 (Kyber512), 54,000 (Kyber768), and 4,000,000 (Kyber1024) inequalities are required for key recovery.

We validated the effectiveness of our attack through experiments on an STM32F4 target board, with introducing the required faults by clock glitches. The experimental results demonstrate that the required number of faulted decapsualtion can be reduced from more than 540, 000 to 380, 000 when dealing with Kyber512. These findings highlight the risks brought by masking when considering fault attacks.

Keywords: Post-quantum cryptography · ML-KEM · CRYSTALS-Kyber · Fault attack · Masking

1 Introduction

The National Institute of Standards and Technology (NIST) has recently made available for public a set of standards for post-quantum cryptography (PQC) algorithms. As these algorithms will be deployed in a multitude of embedded devices across the globe, it is of paramount importance to ensure their resilience against physical attacks. While the security of side-channel attacks has been extensively studied, that of fault attacks on PQC algorithms has received comparatively little attention. It is therefore imperative to evaluate the security of implementations of PQC algorithms to fault attacks.

In this study, we investigate the vulnerability of the CRYSTALS-Kyber (Kyber) algorithm, which serves as the foundation for the Module-Lattice Key Encapsulation Mechanism Standard (ML-KEM), to fault attacks. To achieve chosen-ciphertext attack (CCA) security, the designers of Kyber applied the Fujisaki-Okamoto (FO) transform [12] to a chosen-plaintext attack (CPA) secure public key encryption (PKE) algorithm. During decapsulation, the ciphertext is first decrypted and then re-encrypted. In the event that the re-encrypted ciphertext does not match the original, the decapsulation process fails. From a fault attack perspective, the FO transform can be regarded as a redundancy countermeasure [1] designed to protect against malicious tampering. As a result, traditional fault attack techniques, such as the differential fault attack [3], are not directly applicable to Kyber.

Fault attacks targeting the key generation phase [22][11] or disrupting the FO transform [17][24] have proven effective. Several such attacks have been proposed [21], although they often involve high costs or are challenging to execute in practice. However, even observing whether an injected fault results in a decapsulation failure can benefit the attacker. Indeed, fault attacks have been devised that exploit this concept to achieve key recovery, including the safe-error attack [25][2] and the (statistical) ineffective fault attack [10]. Based on this concept, in 2020, Pessl *et al.* proposed a new fault attack on CCA-secure KEMs [20]. Following this, Hermelink *et al.* introduced a fault-enabled chosen-ciphertext attack on Kyber by combining fault injection with a chosen-ciphertext attack [14]. Their attack requires a single-bit flip and remains effective even with the presence of shuffling countermeasures. Later, Delvaux proposed a more general attack, known as the Roulette attack, along with an efficient solver [9]. Building on these advancements, Hermelink *et al.* [13] incorporated the solver with the lattice reduction framework from [7], reducing the number of inequalities required for successful key recovery.

Masking serves as a generic countermeasure to defend against side-channel attacks. It involves dividing a single intermediate value into multiple random shares, making it challenging to tamper with an intermediate value precisely using fault injection. Consequently, it can effectively defend against the majority of fault attacks. In lattice-based schemes, masking encompasses arithmetic masking, Boolean masking, and the conversion between arithmetic and Boolean masking (A2B and B2A). This increases the complexity of the algorithmic implementation. However, it remains unclear whether this added complexity introduces new vulnerabilities to fault attacks.

Delvaux's work [9] was among the first to explore this issue, focusing on the linear components of masked Kyber, such as polynomial multiplication and the number theoretic transform. He pointed out that the randomness inherent in masking could expedite the Roulette attack. However, more complex components, such as non-linear components like the message decoder were beyond the scope of this study. In 2024, Kundu *et al.* [15] identified an adder-carry leakage in the masked decoder proposed by Oder *et al.* [18], which enables a practical fault attack on masked Kyber. This highlights the need for increased scrutiny of non-linear components in masked implementations. Building on these findings, we aim to investigate one of the most advanced masked schemes of Kyber, which is proposed in [5]. Since attacks on the addition of offset [20] and A2B conversion [15] have been thoroughly investigated, this paper primarily focuses on the less-explored left part of the masked decoder. In essential, the principle contribution of this work can be summarized as follows:

– We conducted an analysis of the effectiveness of fault injection attacks on the masked Kyber scheme, identifying an exploitable fault attack vulnerability. Based on a practical attacker model, we demonstrate how an attacker can derive a system of inequalities that reveal information about the secret key. To the best of our knowledge, this is the first practical fault attack targeting the arbitrary-order masked message decoder in lattice-based schemes.
– By adapting a solver for the system of inequalities and employing two filtering techniques, we found that the secret keys can be recovered with no more than 36,000, 54,000, and 4,000,000 inequalities for Kyber512, Kyber768, and Kyber1024, respectively. As a side contribution, this is also the first evaluation of the cost of fault attack on the masked non-linear components of Kyber768 and Kyber1024.
– Leveraging the implementation characteristics of this masking scheme, we demonstrated that a simple instruction skip fault in the Bitslice process can achieve the desired bit-flipping effect. The feasibility of this fault injection method was validated through inducing clock glitches on a STM32F4 target board, resulting in successful key recovery. This approach demonstrates a significantly higher fault injection success rate compared to previous attacks, consequently necessitating fewer decapsulation attempts for successful key recovery. A detailed comparison is presented in Table 7.

Outline This paper is organized as follows. In Section 2, we present the background knowledge utilized in this paper. Section 3 details our observations and introduces the basic attack method. In Section 4, we describe how to perform the attack on a practical masked implementation. In Section 5, we present practical evidence that supports our claims. Finally, we conclude this papaer and discuss potential countermeasures in Section 6.

2 Preliminaries

2.1 Notations

To simplify notation, we denote the ring of integers modulo q as $\mathbb{Z}_q$ and the ring of $\mathbb{Z}_q[X]/(X^n+1)$ as $\mathcal{R}_q$. Furthermore, we denote by $\mathbb{B}^k$ the set of byte arrays of length k. Regular font letters, such as v, denote elements in $\mathcal{R}_q$, with $v[i]$ representing the i-th coefficient in v. Specially, when we refer to a message $m \in \mathbb{B}^{32}$, we use $m[i]$ to denote the i-th bit in this message. Bold lower-case letters indicate vectors. By default, all vectors are column vectors, and we denote $\mathbf{v}[i]$ the i-th entry of a vector $\mathbf{v}$, and $\mathbf{v}[i,j]$ the j-th coefficient (bit) in its i-th entry when the entry is a polynomial (message), with the index starting at zero. For an element $z \in \mathbb{Q}$ we denote by $\lceil z \rfloor$ to the closest integer with ties being rounded up. z_i denotes the i-th bit in z, while $z^{(i)}$ denotes the i-th share of masked z. The notation $z^{(\cdot)}$ represents all shares of masked z. Additionally, the symbols $z^{(\cdot)A}$ and $z^{(\cdot)B}$ are used to denote arithmetic masking and Boolean masking, respectively. When dealing with polynoimials, the symbol $\circ$ represents vector multiplication between vectors (matrices and vectors).

2.2 Masked Kyber

The Kyber algorithm is constructed from a public-key encryption scheme (KyberPKE) and employs the Fujisaki-Okamoto (FO) transform [12] to convert the PKE into a CCA-secure key encapsulation mechanism (KyberKEM). A detailed specification is provided in Appendix A.

The secret-key-dependent operation in Kyber is the key decapsulation, in which the secret key and the ciphertext are first used to compute a noisy polynomial mp. Subsequently, the decoding function ($\text{Compress}_q(z,1)$) maps each coefficient of the polynomial to a corresponding message bit. As illustrated in Fig. 1, a coefficient close to 0 is decoded as 0, while a coefficient close to $\lceil \frac{q}{2} \rfloor$ is decoded as 1. Following decoding, the recovered message is re-encrypted. If the newly generated ciphertext does not match the original one, a decapsulation failure occurs, and a meaningless output is returned; Otherwise, the correct shared key is established and returned.

Bos *et al.* proposed the first comprehensive implementation of masked Kyber [5]. In their work, they introduced a new masked decoder for message decoding, which is based on a masked bit-sliced binary search. The core part of this decoder is the Compress_q^s function, as shown in Equation 1.

$$\text{Compress}_q^s(z) = \neg z_{11} \oplus (\neg z_{11} \cdot z_{10} \cdot z_9 \cdot (z_8 \oplus (\neg z_8 \cdot z_7))) \tag{1}$$

As illustrated in Algorithm 1, the whole decoding process starts with adding the offset $\left\lfloor \frac{3q}{4} \right\rfloor$ modulo q from the arithmetic shares with a subsequent $\mathsf{A2B}$ conversion to create $\lceil \log_2 q \rceil$-bit Boolean shares of the coefficient. Given these Boolean shares, it then suffices to securely compute whether the masked value is smaller than $\frac{q}{2}$ using Equation 1, where

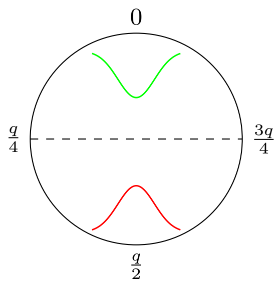

Fig. 1. The original decoding result. The red curve depicts the distribution of polynomial coefficients that are decoded as 1, whereas the green curve depicts the distribution of coefficients decoded as 0.

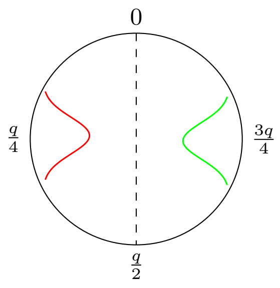

Fig. 2. The decoding result after $+3q/4$. The red curve depicts the distribution of polynomial coefficients decoded as 1, whereas the green curve depicts the distribution of coefficients decoded as 0.

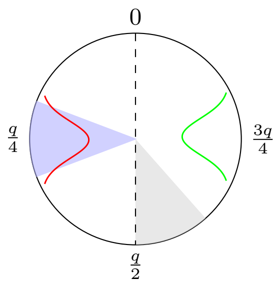

Fig. 3. The shaded regions depict intervals of coefficient that result in decapsulation failures after fault injection. Specifically, the blue regions denote interval containing coefficients exploitable in practical attacks.

$$\text{Compress}_q^s(z) = \begin{cases} 1, & \text{if } z < \dfrac{q}{2} \\ 0, & \text{otherwise} \end{cases}$$

The process correspond to the decoding results in Fig. 2. Following this, a masked binary search is performed, with the $\oplus$ and $\cdot$ operations replaced by their secure counterparts (SecXOR and SecAND). Additionally, the SecREF function is required for fulfilling security properties, which is beyond the scope of this paper. To enhance efficiency, the Boolean shares of the polynomial are converted into a bit-sliced representation, enabling the compress function to be executed in parallel. This bit slicing transformation is crucial to the performance of the masking decoder.

3 Generic Attack Description

In this section, we present our primary observation, which allows for a practical fault attack on the masking scheme proposed in [5]. The attack vulnerability arises from the intricate implementation of the masked message decoder, rather than from the original components of the plain Kyber. For the sake of simplicity, we will initially describe our attack in a non-masked manner, disregarding the obfuscation caused by masking and focus solely on the description provided in Equation 1. Additionally, unless otherwise specified, all related parameter sets refer to Kyber512. The detailed attack on a specific masked implementation will

Algorithm 1 Masked Decoder $\text{Compress}_q(z, 1) = \text{Compress}^s_q\left(z + \left\lfloor \frac{3q}{4} \right\rfloor \mod q\right)$

```
Input: $a^{(\cdot)A}$, $a \in \mathbb{Z}_q[X]$.
Output: $m'^{(\cdot)B}$, $m' = \text{Compress}_q(a, 1) \in \mathbb{Z}_{2^{256}}$.
 1: for $i \leftarrow 0$ to $n-1$ do
 2:     $a_i^{(0)A} = a_i^{(0)A} + \left\lfloor \frac{3q}{4} \right\rfloor \mod q$
 3:     $a_i^{(\cdot)B} = \text{A2B}(a_i^{(\cdot)A})$
 4: end for
 5: $z^{(\cdot)B} = \text{Bitslice}(a^{(\cdot)B})$
 6: $m'^{(\cdot)B} = \text{SecAND}(\text{SecREF}(\neg z_8^{(\cdot)B}), z_7^{(\cdot)B})$
 7: $m'^{(\cdot)B} = \text{SecREF}(\text{SecXOR}(m'^{(\cdot)B}, z_8^{(\cdot)B}))$
 8: $m'^{(\cdot)B} = \text{SecAND}(m'^{(\cdot)B}, z_9^{(\cdot)B})$
 9: $m'^{(\cdot)B} = \text{SecAND}(m'^{(\cdot)B}, z_{10}^{(\cdot)B})$
10: $m'^{(\cdot)B} = \text{SecAND}(m'^{(\cdot)B}, \neg z_{11}^{(\cdot)B})$
11: $m'^{(\cdot)B} = \text{SecXOR}(m'^{(\cdot)B}, \neg z_{11}^{(\cdot)B})$
12: return $m'^{(\cdot)B}$
```

be discussed in the following sections. Our attacker model is described as follows, which is a reasonable setting in a practical attack scenario.

Attacker Model It is assumed that the attacker has access to the public key and is able to perform an arbitrary number of encapsulations using suitable messages. The attacker then transmits the resulting ciphertext from encapsulation to the target device and introduces a fault during the decapsulation process. The fault injection causes a bit flip in a decoded coefficient z, either before or during the decoding process. Ultimately, the attacker can observe whether a decapsulation failure occurs in accordance with the equality of the share keys derived from encapsulation and fault injected decapsulation.

Each faulted decapsulation result reveals partial information about the secret key, which is represented as an inequality involving the secret key as the unknown variable. By repeating this process and collecting sufficient inequalities, the attacker can solve the system of inequalities and thereby recover the full secret key.

3.1 Fault Effectiveness in Masked Decoder

Since the masked message decoder relies on bit-wise operations, introducing a bit fault may be easier compared to other fault types. Therefore, we first consider the impact of different bit flips. If a bit flip in z causes a flip in a decoded message bit d, an attacker will observe a decapsulation failure. Our primary objective is to determine whether flipping a specific bit in z will lead to a decapsulation failure and, more importantly, to assess what information the decapsulation result might reveal to the attacker.

As previously stated in Equation 1, the binary search-based decoding process only involves the five bits $z_7 \dots z_{11}$, and thus we will omit the analysis of the remain lower bits in z. Additionally, when $z_{11} = 1$, the decoding result is always

Table 1. The possible intervals in which flipping z_{10} causes a decapsulation failure.

z_{10}	z_9	z_8	z_7	Interval of z	d
1	1	0	1	[1664, 2048)	$0 \rightarrow 1$
1	1	1	0	[1792, 2048)	
0	1	0	1	[640, 1024)	$1 \rightarrow 0$
0	1	1	0	[768, 1024)	

Table 2. The interval information of z provided by the faulted decapsulation results.

	Decapsulation Failure	Decapsulation Success
z_{11}	$[0, 1792) \cup [2048, 3329)$	$[1792, 2048)$
z_{10}	$[640, 1024) \cup [1664, 2048)$	$[0, 640) \cup [1024, 1664)$
z_9	$[1152, 2048)$	$[0, 1152)$
z_8	$[1664, 1792)$	$[0, 1664) \cup [1792, 2048)$
z_7	$[1792, 1920)$	$[0, 1792) \cup [1920, 2048)$

fixed at 0, regardless of the values of the other bits, as shown in Equation 2. Consequently, when focusing on the lower bits, we omit the interval $[2048, 3329)$, as it corresponds to $z_{11} = 1$.

$$d = 0 \oplus (0 \cdot z_{10} \cdot z_9 \cdot (z_8 \oplus (\neg z_8 \cdot z_7))) = 0 \tag{2}$$

We use z_{10} as an example to illustrate our analysis. Since we are only considering the case where $z_{11} = 0$, we can update Equation 1 to reflect this in Equation 3. The equation can be divided into two parts: one involving z_{10} only and the other consisting of the remaining bits. If either of the two parts equals to 0, the decoded message bit will be fixed to 0. This means that flipping z_{10} only becomes significant if the condition $z_9 \cdot (z_8 \oplus (\neg z_8 \cdot z_7)) \neq 0$ is satisfied. We recursively analyze the second part of the equation until all possible cases are considered. The results of this analysis are summarized in Table 1, where it is shown that flipping z_{10} will cause a flip in d and lead to decapsulation failure when $z \in [640, 1024) \cup [1664, 2048)$.

$$\text{Compress}_q^s(z) = \neg(z_{10} \cdot z_9 \cdot (z_8 \oplus (\neg z_8 \cdot z_7))), z < 2048 \tag{3}$$

We extend our recursive analysis to all five bits from z_7 to z_{11}, and the final results are presented in Table 2. The analysis reveals that if an attacker can flip any of the bits involved in the masked decoder, then observing the decapsulation result provides interval information about the decoded coefficient. This key insight allows the attacker to narrow down the possible values of the decoded coefficient, thereby facilitating the recovery of the secret key.

Nevertheless, determining whether this interval information can be exploited by an attacker requires further investigation. In accordance with the Kyber specification presented in Appendix A, the decoded polynomial, denoted as *mp*, is computed according to Equation 4. Here, μ denotes the polynomial derived from the encoded message, and we define δ as the noise term to be eliminated during the decoding process. Let $\gamma = \Pr[\|\sigma\|_\infty \geq \lceil q/4 \rfloor]$. The design of Kyber [4] aims to achieve a security level of $1 - \gamma$ by ensuring elements of δ follow a specific probability distribution, thereby satisfying the necessary security properties. Additionally, the Kyber team has provided a Python script[4] for evaluating the probability distribution of δ.

$$\begin{aligned} mp &= v + \Delta v - (\mathbf{u} + \Delta \mathbf{u}) \circ \mathbf{s} \\ &= \mathbf{t} \circ \mathbf{r} + e_2 + \mu + \Delta v - (\mathbf{A} \circ \mathbf{r} + \mathbf{e_1} + \Delta \mathbf{u}) \circ \mathbf{s} \\ &= \mathbf{r} \circ \mathbf{e} - (\mathbf{e}_1 + \Delta \mathbf{u}) \circ \mathbf{s} + e_2 + \Delta v + \mu \\ &= \delta + \mu \end{aligned} \tag{4}$$

Since the input coefficient z for Compress_q^s is equal to either $\lceil q/4 \rfloor + \delta$ or $\lceil 3q/4 \rfloor + \delta$, the script can also be used to evaluate the probability distribution of decoded coefficients. To illustrate, in the case of flipping z_8, a decapsulation failure occurs only if $z \in [1664, 1792)$. Accordingly, the probability of a noisy coefficient occurring within this interval is $2^{-103.9}$. This indicates that the probability of flipping z_8 and resulting in a decapsulation failure is essentially zero. Similarly, the same reasoning applies to other bits.

In accordance with the aforementioned guideline, selecting z_{10} for fault injection proves to be the optimal choice. When an attacker flips the z_{10} bit from 0 to 1, the probability that z falls into a interval resulting in decapsulation failure is $1 - 2^{-6.8}$ (approximately 99.1%), which is within an acceptable range for practical attacks. On the other hand, the opposite flip direction, corresponding to the interval $[1664, 2048)$, has a negligible probability of $2^{-39.8}$, making it impractical for exploitation in a real-world attack scenario. Consequently, when targeting the z_{10} bit for a fault attack, the entire interval of z can be divided into three distinct regions, as illustrated in Fig. 3.

If the value of z falls into the blank area in Fig. 3, a bit flip happened to z_{10} does not affect the decapsulation result, providing no useful information for the attack. However, when a decoded coefficient lies in the interval $[640, 1024)$ (the blue region) and z_{10} is flipped by fault injection, the attacker will observe a decapsulation failure. Similarly, the interval $[1664, 2048)$ (the gray region) has the same effect. Although a coefficient lies in $[1664, 2048)$ occurs with very low probability, it may still introduce some noise to the attack. Since the attacker has the ability to control the message bit corresponding to z during the encapsulation phase, they can ensure that $z < \frac{q}{2}$ by fixing the target message bit at 1. Furthermore, Kyber has an inherently low failure probability, as indicated in Table 8, which can be neglected in this attack due to its negligible occurrence. This leads to the following conclusion: during the decapsulation of a filtered ciphertext, if a bit flip occurs in z_{10} and the attacker observes a decapsulation failure, they can infer that $z \in [640, 1024)$. Otherwise, $z \in [0, 640) \cup [1024, 1664)$.

[4] https://github.com/pq-crystals/security-estimates

The remainder of this section will demonstrate how key recovery can be achieved based on this observation.

3.2 Key Recovery Attack

In Equation 4, $\Delta\mathbf{u}$ and Δv represent the losses introduced by the lossy compression of $\mathbf{u}$ and v, respectively. The value of Δv is computed as

$$\Delta v = \text{Decompress}(\text{Compress}(v, d_v), d_v) - v,$$

and $\Delta\mathbf{u}$ is determined analogously. Consequently, both Δv and $\Delta\mathbf{u}$ are publicly accessible. Furthermore, the variables $\mathbf{e}_1$, $\mathbf{r}$, e_2, and μ are involved in the encapsulation process and are therefore under the attacker's control. The only unknowns are $\mathbf{s}$ and $\mathbf{e}$, which are secret vectors generated during the key generation phase.

Following the description of the masked decoder, we set $z = mp[i] + \lfloor 3q/4 \rfloor \bmod q$ as the input to Compress_q^s. Thus, after a faulted decapsulation with a flip in z_{10}, if a decapsulation failure is observed, the attacker can infer that $z \in [640, 1024)$, which is equivalent to $\delta[i] \in [-192, 192)$. If a decapsulation success is observed, then $\delta[i] < -192$ or $\delta[i] \geq 192$. After performing ω faulted decapsultions, the attacker can obtain ω inequalities, represented as Equation 5. Here, $\mathbf{M}$ and $\mathbf{b}$ represent the public information from the ω inequilities, while $\mathbf{x}$ represents the unknown vector of $\mathbf{s}$ and $\mathbf{e}$, which needs to be solved.This vector contains $2nk$ unknown coefficients, with each coefficient having $2\eta_1 + 1$ possible values. For simplicity, we refer to inequalities featuring the $\notin$ symbol as negative inequalities and those featuring the $\in$ symbol as positive inequalities. Throughout the remainder of this paper, we denote $\psi = 2nk$ and set $i = 0$.

$$\mathbf{Mx} + \mathbf{b} = \begin{pmatrix} (\mathbf{r})_{(0)}, -(\mathbf{e}_1 + \Delta\mathbf{u})_{(0)} \\ \cdots \\ (\mathbf{r})_{(\omega-1)}, -(\mathbf{e}_1 + \Delta\mathbf{u})_{(\omega-1)} \end{pmatrix} \begin{pmatrix} \mathbf{e} \\ \mathbf{s} \end{pmatrix} + e_2 + \Delta v \underset{\notin}{\in} [-192, 192) \quad (5)$$

Given the large number of inequalities and the inherent error rate in fault attacks, directly solving such a system of inequalities is impractical. To address this challenge, a series of solvers have been proposed, and the underlying approach is similar across these solvers. First, the distribution of all secret coefficients is initialized based on their sampling distribution. For instance, in Kyber, the coefficients in $\mathbf{s}$ and $\mathbf{e}$ follow a centered binomal distribution parameterized by η_1. Next, each coefficient iteratively updates its distribution based on the information provided by the system of inequalities. For the i-th inequality, which corresponds to a decryption failure, the probability that the k-th candidate of the j-th unknown coefficient is the correct guess is updated as shown in Equation 6. This equation captures how the probability distribution evolves over iterations by factoring in contributions from the current inequality and other coefficients. The iterative process continues until a termination condition is satisfied. At that point, each coefficient selects the candidate with the highest probability as the final prediction for the secret key.

$$\mathsf{P}[i,j,k] = \Pr\left(-192 \le \mathbf{M}[i,j](k-\eta_1) + \left(\sum_{j' \in [0,\psi-1]\setminus\{j\}} \mathbf{M}[i,j']\mathbf{x}[j']\right) + \mathbf{b}[i] < 192\right) \tag{6}$$

To implement such a solver, the belief propagation (BP)-based approach can be utilized, as demonstrated in [14]. The main bottleneck of this solver lies in the convolution operation, which involves computing all possible combinations of $x[j']$ for $j' \neq j$. To accelerate the process, previous work [20] proposed employing the Fast Fourier Transform (FFT) to optimize these convolutions, significantly reducing the time required to perform the updates. Additionally, a binary tree algorithm was introduced to avoid inefficient re-computations. More recently, a simplified method was presented in [9], which further reduces computational overhead. This method leverages the Central Limit Theorem (CLT) to approximate the sum of distributions using a normal distribution X. The mean of X is given by

$$\mu = \mathbf{M}[i,j](k-\eta_1) + \left(\sum_{j' \in [0,\psi-1]\setminus\{j\}} \mathbf{M}[i,j']\mathsf{E}[\mathbf{x}[j']]\right) + \mathbf{b}[i],$$

and the standard deviation is calculated as

$$\sigma = \sqrt{\sum_{j' \in [0,\psi-1]} \mathbf{M}[i,j']^2 \mathsf{Var}[\mathbf{x}[j']]}.$$

To convert it into a standard normal distribution, we perform the normalization and get the standard normal distribution Z. Therefore, by revisiting the solver, we can approximate Equation 6 as Equation 7.

$$\mathsf{P}[i,j,k] \approx \Pr\left(\frac{-192-\mu}{\sigma} \le Z < \frac{192-\mu}{\sigma}\right) \tag{7}$$

This allows for rapid calculation of the probabilities using the cumulative distribution function (CDF) of the standard normal distribution, as shown in Equation 8. By applying this approximation, the solving process is significantly accelerated, reducing the time required for handreds times.

$$\mathsf{P}[i,j,k] \approx F_{norm}(\frac{192-\mu}{\sigma}) - F_{norm}(\frac{-192-\mu}{\sigma}) \tag{8}$$

In our attack, we adopt this simplified solver approach by adapting the probability calculation according to Equation 7 and Equation 8, thus tailoring it specifically to the inequalities derived in our attack. Moreover, by leveraging the CLT approximation and PMF calculations, this solver efficiently manages the substantial volume of inequalities generated throughout the attack, thereby enabling a feasible key recovery process even under constrained computational resources.

Table 3. The maximum and minimum values of Δv and e_2.

	Δv		e_2	
	#min	#max	#min	#max
Kyber512/768	-104	104	-2	2
Kyber1024	-52	52	-2	2

3.3 Boosting the Attack with Inequalities Filter

The attack proposed in [9] derives information from faulted decapsulations in the form of inequalities indicating either $\delta \geq 0$ or $\delta < 0$. Since the mean of the distribution of δ is 0, the proportions of positive and negative inequalities are balanced, approximately 50%:50%. In contrast, the inequalities obtained in our attack provide relatively unbalanced interval information. Specifically, for randomly generated ciphertexts, the obtained negative inequalities account for only $2^{-6.8} \approx 0.9\%$ in Kyber512, which presents a significant disadvantage for our solver. To address this, we adapt two filtering techniques to enhance the effectiveness of our attack. For comparison, the updated proportions after applying the filters are presented in Table 4.

Filter1: Removing low-contribution inequalities offline. In most cases, a noise coefficient $\delta[i] \in [-192, 192)$. Consequently, for many positive inequalities, most or all possible values of the secret coefficients satisfy the inequality. This renders these inequalities largely ineffective for distinguishing the correct candidates. We categorize such inequalities as low-contribution inequalities. To enhance the attack, it is crucial to identify high-contribution inequalities. According to the Central Limit Theorem, $\delta[i]$ is distributed approximately a normal distribution with mean $(\Delta_v + e_2)[i]$. Consequently, selecting ciphertexts where $(\Delta v + e_2)[i]$ approaches the boundary values ± 192 results in more incorrect secret coefficient candidates being excluded for the corresponding inequality.

The value of e_2 is sampled from a distribution parameterized by η_2, while the value of Δv are associated with the parameter d_v. The interval information for these two values is provided in Table 3. Without loss of generality, we choose to use the ciphertext of encapsulation where $|(\Delta v + e_2)[i]| \geq 94$ for Kyber512/768 and $|(\Delta v + e_2)[i]| \geq 42$ for Kyber1024.

This filtering technique allows the attacker to focus on ciphertexts more likely to produce high-contribution inequalities, significantly enhancing the effectiveness of our attack. Moreover, since suitable ciphertexts can be selected offline, this approach does not increase the overall attack cost.

We can infer that the required number of inequalities to solve the secret key for Kyber1024 will increase at a rate faster than what would be expected based solely on the increase in security level. This is due to the more biased values present in Kyber1024. A detailed analysis of this observation will be provided in the following subsection.

Table 4. The proportions of decapsulation success and failure.

	Without filter		With filter1		With filter1 and filter2	
	#fail	#success	#fail	#success	#fail	#success
Kyber512	99.1%	0.9%	95.5%	4.5%	$(95.5\% - \alpha)/(1-\alpha)$	$4.5\%/(1-\alpha)$
Kyber768	99.41%	0.59%	97%	3%	$(97\% - \alpha)/(1-\alpha)$	$3\%/(1-\alpha)$
Kyber1024	99.931%	0.069%	99.76%	0.24%	$(99.76\% - \alpha)/(1-\alpha)$	$0.24\%/(1-\alpha)$

Filter2: Increase the Proportion of Negative Inequalities. In our attack, we deal with a highly unbalanced system of inequalities, where the number of negative inequalities is significantly less than the positive ones. We have observed that this imbalance can cause the solver to converge prematurely. Initially, for Kyber512, the proportion of negative to positive inequalities was approx 0.9 : 99.1. With the filter1 applied, this proportion can be improved to some extend. There is no theoretical method to calculate the new proportion. We provide an experimental result in Table 4. However, this improvement is still insufficient for Kyber1024. To further address this issue, we revisit another filtering technique proposed in [15], called *rejection sampling*. In this approach, a random proportion α of the inequalities is rejected, but only from those generated in cases of positive inequalities. This selective rejection helps increase the proportion of negative inequalities, balancing the system and improving the performance of the solver, this improvement can be seen in Table 4. For example, with $\alpha = 94\%$, the proportion of inequalities from Kyber1024 can be brought to the same level as Kyber512 only using filter1.

4 Practical Attack on Masked Kyber

The attack described in Section 3 omits specific details of the masked implementation to provide a more intuitive understanding of our approach. In this section, we will delve into how the attack can be successfully executed in a practical masked implementation. Considering our attacker model, fault injections can be categorized into the following types:

- **Exact Fault**: The fault injection achieves the desired effect. Specifically, a noisy coefficient in the range $[640, 1024)$ results in a decapsulation failure; otherwise, decapsulation success is observed.
- **Invalid Fault**: The fault injection does not produce the expected outcome. For example, a noisy coefficient in the range $[640, 1024)$ does not lead to a decapsulation failure. This can be caused by either a failed fault injection or interference from the masking, which will be clarified later in this section.
- **Reset Fault**: The fault injection has an unintended consequence, disrupting the control flow and rendering the process unusable.
- **Unintended Fault**: The fault injection causes an unexpected result, such as a noisy coefficient outsides the range $[640, 1024)$ that still leads to a decapsulation failure.

4.1 Bit-Flip in Masking Implementation

First, considering the masking, an intermediate value is divided into $t+1$ shares. Before message decoding, these shares are transformed into Boolean masking using the A2B transformation, as outlined in Algorithm 1. For a Boolean-masked coefficient, if any share of a bit in z is flipped, such as $z_{10}^{(0)}$, the unmasked value of z_{10} will also be flipped.

This makes the bit-flip fault model ideal for our attack, which has already been applied in the attack proposed in [14]. However, executing precise bit flips in a practical setting is challenging. The stuck-at fault model used in [15] can also be applied to our attack. If $z_{10}^{(0)}$ is stuck at 0 or 1, due to the randomness of masking, on average, two decapsulations of the same ciphertext will yield a valid flip of the unmasked bit. By decapsulating a ciphertext β times, and with an appropriate choice of β, the effectiveness of a stuck-at fault approaches that of an ideal bit-flip fault, though it requires several times the number of additional decapsulation attempts.

The bit-flip and stuck-at faults can be injected over a relatively long time window, as shown in Table 5. Nonetheless, it is important to emphasize that inducing a single-bit fault remains highly challenging for an attacker.

4.2 Bit-Flip Based on Instruction Skip

The use of bitwise operations in the masked scheme has enabled the development of a more relaxed fault injection method for achieving the desired fault model. In [5], the authors stress the necessity of transforming the Boolean shares of the polynomial to a bit-sliced representation prior to compute the compression function. The bit-sliced representation allows parallel accelaration in an otherwise serial software implementation, offering a performance improvement concerned about the word size of the target platform. Given that only the five most significant bits will be used in subsequent operations, it is prudent to discard the lower seven bits in order to reduce the cost.

Although the implementation of BitSlice strongly depends on the capabilities of the target platform, the fundamental logic remains consistent. A basic description of the naive BitSlice process is provided in Algorithm 2, where the five most significant bits of l coefficients are stored in five registers in the bit-sliced representation. Notably, a critical step involves assigning a single bit of a coefficient to the target register. If an attacker manages to skip this step, the result bit will be fixed at 0, effectively creating a stuck-at-0 fault. Since the implementation is masked, the final outcome will appear as a random bit flip on the unmasked value with a probability of 50%. As we can perform β times fault injected decapsulations of the same ciphertext, once a decapsulation happens, then we can classify the inequality from this ciphertext as a positive inequality. Otherwise, all β decapsulations succeed, then a negative inequality generated. By selecting an appropriate β, we can minimize the influence of masking and invalid fault on our attack, and reduce the error rate of negative inequalities to nearly 0.

Algorithm 2 BitSlice

```
Input: $a$, a polynomial
Input: $l$, word size of target platform
Output: $b$, the bit-sliced representation of $a$
 1: for $i \leftarrow 0$ to $l-1$ do
 2:     $b[i] = 0$
 3: end for
 4: for $i \leftarrow 0$ to $4$ do
 5:     for $j \leftarrow 0$ to $l-1$ do
 6:         $bit = (a[j] \gg (i+7)) \& 1$
 7:         $bit = bit \ll j$
 8:         $b[i] = b[i] \mid bit$
 9:     end for
10: end for
```

Table 5. Available fault injection types for our attack.

Fault model	Injection Target
Bit-Flip Stuck-at 0/1	A2B Bitslice SecAND load/store
Instruction Skip	Bitslice

4.3 Other Feasible Fault Models

In addition to the exact instruction skip, other operations can also be leveraged for the attack. For instance, if the right operation ($\gg$) operation is skipped, then $z_0^{(0)}$ will replace $z_{10}^{(0)}$, resulting in a randon bit fault. Similarly, if the left shift ($\ll$) operation is skipped, $a[j]_{10}^{(0)}$ will be set as 0 and $a[0]_{10}^{(0)}$ will incur a random fault.

Moreover, if the attacker have the capability to perform a single-bit fault injection, such as the stuck-at-0/1 fault used in [15], a more wide range of fault injection becomes available for our attack. Any operation involving z_{10} can serve as a target. A summary is provided in Table 5, demonstrating that key recovery can still be accomplished even without bit-sliced optimization. However, the use of bit slicing significantly weakens the requirements for the attacker.

5 Experiments

In this section, we first evaluate both the correctness and the performance of our attack based on the results of software simulation experiments. Subsequently, the feasibility of the attack was validated in a practical attack scenario by executing the attack on a STM32F405 target board, which utilizes clock glitches to induce instruction skip faults.

5.1 Experiments Setup

We employed the Chipwhisperer suite [19] to carry out our fault attack. The suite comprises an STM32F405 target board, which incorporates an ARM Cortex-M4 core, in conjunction with a Chipwhisperer-Lite control board. The Chipwhisperer-Lite provides a 28MHz clock to the target board, thus it is also employed to generate clock glitches for fault injection.

The STM32F405 board serves as the victim device. The device receives ciphertext, performs decapsulation with the secret key, and subsequently returns a shared key. Since the implementation in [5] is not open-source and the fault attack only targets the message decoding process, the firmware running on the board is based on Kyber's reference implementation[5], equipped with the masked decoder described in [5]. In this implementation, the BitSlice process is a proof-of-concept implementation based on the naive approach described in Algorithm 2. The code was compiled using the *GNU Arm GCC Compiler 10.3.1* with *-O3* optimization.

When solving the obtained system of inequalities, we primarily used the simplified solver from [9]. Additionally, we also evaluated our attack using the BP-based solver from [13] for comparison. Both solvers were adapted from the source code provided in the respective original papers to suit our attack[6][7]. Empirically, we set the maximum number of iterations for the solvers to 8 and used the average number of recovered coefficients over 10 runs as the final result.

5.2 Evaluation of the Proposed Attack

First, we verify the correctness of our attack on Kyber512, which is expected to the easiest to break. The results, shown in Fig. 4, demonstrate that our attack is capable of recovering the secret coefficients, and the application of filter1 significantly reduces the required number of inequalities. With filter1 is applied, only 36, 000 inequalities are required for a full key recovery. When employing filter2 with $\alpha = 0.5$, we observe an improvement in the proportion of negative inequalities. However, this leads to a decrease in the success rate due to the reduced number of inequalities available for the solver.

Additionally, since there may be an error rate in fault attacks, we conducted another experiment to evaluate the error-tolerance ability of our attack. We observed that if β is not sufficiently large, some positive inequalities may be mistakenly identified as negtive cases. To simulate this scenario, we intentionally corrupted a portion of the positive inequalities within the collected system of inequalities, thereby introducing an error rate in the negative inequalities. The results, shown in Fig. 5, indicate that our attack can tolerate an error rate of up to 30%, with only a reasonable increase in the required number of inequalities.

The results for all three security levels are provided in Fig. 6. The attack difficulty for Kyber768 is comparable to that of Kyber512, which aligns with our analysis. As expected, recovering the secret coefficients for Kyber1024 is significantly harder than for the previous two levels. Based on the analysis in Section 3, we identified that applying filter2 with $\alpha = 0.94$ yields a system of

[5] https://github.com/pq-crystals/kyber
[6] https://github.com/Crypto-TII/roulette
[7] https://github.com/juliusjh/improved_decryption_error_recovery

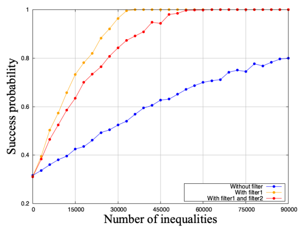

Fig. 4. Solving filtered and unfiltered inequalities for Kyber512.

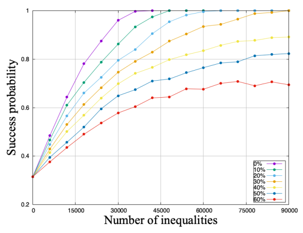

Fig. 5. Solving corrupred inequalities for Kyber512.

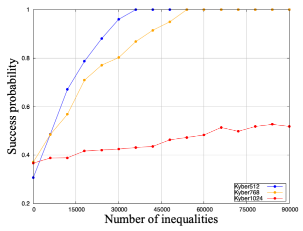

Fig. 6. Solving filtered inequalities for all three security levels.

inequalities with a relatively acceptable ratio of negative to positive inequalities for Kyber1024. We conducted experiments to determine the number of inequalities required for full key recovery, and the results are shown in Fig. 7. The results indicate that 4,000,000 inequalities need to be collected, and the solver must solve a system of 240,000 inequalities for full key recovery.

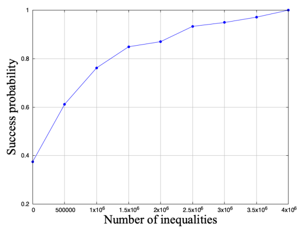

Fig. 7. Solving filtered inequalities for Kyber1024 with $\alpha = 0.94$.

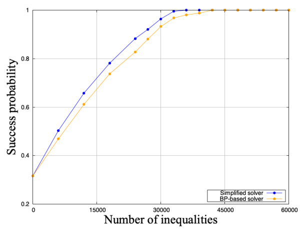

Fig. 8. Comparison between the simplified solver and BP-based solver.

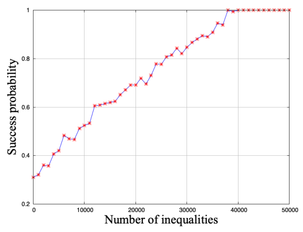

Fig. 9. The key-recovery results of practical experiments.

In [13], Hermelink *et al.* highlighted that the BP-based solver demonstrates an advantage in terms of the number of inequalities when applied to traditional attacks involving balanced inequalities. For comparison, we conducted an experiment to solve our system of inequalities using both solvers, with the results presented in Fig. 8. The findings indicate that, in our attack, the BP-based solver does not exhibit a discernible advantage. Given its higher requirements for runtime and memory, there is insufficient motivation to employ the BP-based solver in our attack.

Table 6. The intervals during parameters scanning.

offset	width	ext_offset	repeat
$[-20, 20]$	$[1, 20]$	$[1, 50]$	1

5.3 Fault Profiling

To achieve an effective fault injection, selecting the correct fault injection parameter set is essential. For clock glitch-based fault injection, the main parameters involve determining when to insert the glitch (ext_offset) and what type of glitch to apply (offset and width). In this subsection, we detail the fault profiling process in our experiments.

A naive method to locate the target operation would be to scan through the entire decapsulation process from the beginning. However, this approach is highly time-consuming. Instead, we found that the start of the BitSlice process can be coarsely identified using power side-channel information. Since we are targeting the first coefficient, only a small portion of the process needs to be scanned. For simplicity, in our experiments, we placed a trigger signal at the beginning of the Bitslice process. It is worth noting, however, that the target operation remains identifiable solely through power trace analysis. This implies that the absence of a trigger signal would not impede attacks in real-world scenarios.

Even with precise identification of the target operation, there may still be a probability of fault injection failure, as the analysis in Section 4. Therefore, a critical aspect of the attack is how to handle error inequalities caused by imperfect fault injections.

Since the attacker can only observe whether decapsulation succeeds or fails, it is not possible to directly distinguish between an exact injection and an unintended injection in the case of a decapsulation failure. In the event of a destroyed injection, the attacker can simply retry the decapsulation with the same ciphertext. Similar to the strategy employed in [15], multiple faulted decapsulations can be performed on a single ciphertext to mitigate the issue of invalid injections. Fortunately, the solver used in our attack demonstrates a high tolerance for errors, minimizing concerns about destroyed-injection.

To determine suitable parameters for fault injection, we performed a scan across the intervals shown in Table 6. The offset and width represent the timing and duration of the glitch within a single clock cycle. Choosing smaller values for these parameters minimizes their impact on execution and reduces the likelihood of unintended faults. Similarly, the repeat parameter, which controls the number of glitches inserted during a single decapsulation, is fixed at 1 to further reduce unintended faults. As for the ext_offset, which determines how many clock cycles after the trigger signal the glitch is inserted, we empirically set the maximum scan value to 50 in our experiment. After the scanning process, the number of decapsulations that returned an unexpected share key from fault injection with varying ext_offset values is shown in Fig. 10. We observed that flips mainly occurred at ext_offset 28, 34, and 20. However, there is also the possiblility the some flips were caused by unintended faults. To filter this parameter, we propose the following method: for each candidate ext_offset, we use a parameter pair of offset and width that enables flips, then collect 500 inequalities through our fault attack. Next, we check whether the number of decapsulation successes aligns with or close to the theoretical proportion. Finally, we found that ext_offsets 28 and 34 did not result in any decapsulation successes, while ext_offset 20 showed the correct proportion. Thus, we selected ext_offset = 20 for our attack configuration.

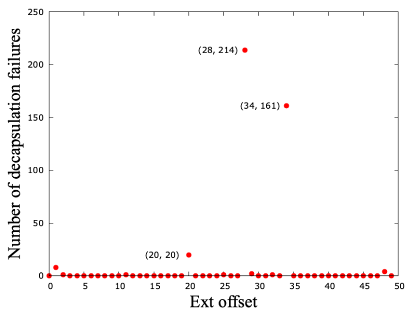

Fig. 10. The number of decapsulation failures with different ext_offset.

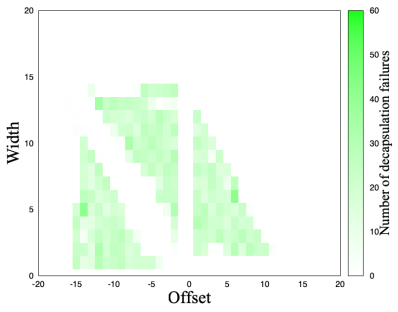

Fig. 11. The number of decapsulation failures with different (offset, width).

Additionally, to minimize the number of invalid injections and thereby lower the total number of decapsulations required for our attack, we fixed ext_offset at 20 and conducted 100 faulted decapsulations for each pair of offset and width. The resulting number of decapsulation failures is presented in Fig. 11.

5.4 Inequalities Acquirements and Key Recovery

Following the fault profiling phase, we conducted our attack using the predetermined parameters. Initially, we collected a total of 50,000 inequalities. To ensure the efficacy of the fault injections and minimize the occurrence of invalid injections, each decapsulation attempt was repeated 10 times. Among the inequalities, 28 negative inequalities are identified as error, indicating an error rate of 6.2%, while no erroneous positive inequalities were observed. We then applied our solver to recover the entire secret key. As illustrated in Fig. 9, we found that with at least 38,000 inequalities, we were able to recover the entire secret key for Kyber512. This result aligns with our simulation experiments, confirming the validity of our attack approach. This demonstrates that even with fewer attempts, our attack remains highly effective, significantly reducing the overall attack cost while maintaining high success rates.

5.5 Comparison

The simulation and practical experimental results indicate that the proposed attack is capable of successfully recovering the secret key of Kyber. A comprehensive analysis and comparison are presented in Table 7. The number of inequalities represents the results from simulation experiments under the assumption of perfect fault injection, while β indicates the required number of decapsulation attempts to perform a successful attack in the practical experiments.

In comparison to the attack described in [9], our attack targets the more complex part of the masked Kyber implementation, specifically the message decoder. The faulted decapsulation results produce an unbalanced system of inequalities, providing less information about the secret key compared to the approach in [9]. Consequently, our attack requires several times more inequalities theoretically. However, the attack in [9] relies on random faults in the polynomial coefficients, necessitating a larger number of decapsulation attempts of a ciphertext to generate an inequality (denoted by β) even with high-precision fault injection.

Specifically, to approach the error rate in our attack with $\beta = 10$ (approximately 6.2%), their attack would require $\beta > 100$ (see Figure 7 in [9]). Thus, in practical scenarios, their attack requires significantly more faulted decapsulation attempts than ours. For instance, their practical experiments indicate that with $\beta \geq 20$, the error rate reaches approximately 50%. Under these conditions, successful key recovery requires 27,000 inequalities, corresponding to a total of 540,000 decapsulation attempts. In contrast, our attack with $\beta = 10$ requires only 38,000 inequalities to recover the secret key, corresponding to 380,000 decapsulation attempts, representing a reduction of approximately 30%.

Furthermore, the attack in [9] necessitates the manipulation of a compressed ciphertext coefficient of an otherwise correctly computed encapsulation. Therefore, the attack can be effectively mitigated by countermeasures such as the message polynomial sanity check proposed in [21]. In contrast, our attack does not rely on manipulated ciphertexts (MC), making it considerably harder to defend.

When compared to the attack described in [15], our approach requires slightly fewer inequalities. This is because a single faulted decapsulation in our attack provides boundary information on both sides, whereas the attack in [15] only obtains one boundary. Notably, our attack only requires a clock glitch to perform an instruction skip, while the attack in [15] requires a single-bit stuck-at fault using expensive electromagnetic pulse technique to inject fault with high precision for injection time and location. Moreover, our fault attack demonstrates a significantly higher success rate, allowing us to use a much smaller β, representing fewer faulted decapsulation attempts. Additionally, we are the first to evaluate the impact of this type of attack on Kyber768 and Kyber1024, discovering notable differences when handling Kyber1024 and providing an evaluation of the cost of recovering the full secret key for Kyber1024.

Table 7. Comparison between this work and previous attacks.

	Type of Inequalities	**Security Level**	**No. Inequalities**	β	**Type of Faults**	**MC Req.**
		Kyber512	36,000			
This work	$\delta \substack{\in \\ \notin} [-192, 192)$	Kyber768	54,000	≥ 10	Clock glitch	✗
		Kyber1024	4,000,000			
[15]	$\delta \substack{\geq \\ <} -192$	Kyber512	60,000	≥ 180	EM pulse	✗
		Kyber512	8,500			
[9]	$\delta \substack{\geq \\ <} 0$	Kyber768	9,400	> 100	Clock glitch	✔
		Kyber1024	12,000			

6 Conclusion

In this paper, we propose a novel fault attack against masked Kyber. By employing an instruction skip fault, an attacker can successfully recover the full secret key of a practical masked Kyber implementation. The effectiveness of our attack stems from the structure of the masked message decoder introduced in [5], further enhanced by bit slicing-optimizated implementation of the decoder. These results underscore the vulnerability of masked implementations to fault attacks. The message decoder can also be extended to other lattice-based schemes, making our attack a broader threat to lattice-based cryptosystems. Notably, both the attacks on non-linear components of masked Kyber, this work and the one described in [15], are based on single-bit faults, demonstrating that the use of bit slicing can be advantageous for fault attackers.

To defend against this attack, several countermeasures can be considered. Mishra et al. present a probabilistic method [16] based on rejection sampling to mitigate this type of attack. However, integrating this approach with a masked implementation poses challenges, as it requires 2 to 3 retries even for a non-faulted decapsulation. In 2022, Coron et al. proposed a new message decoder based on a table-based conversion algorithm [6]. After thorough analysis, we have not identified any similar vulnerabilities in this scheme. However, this method introduces a higher probability of decryption failure, which could be exploited by boosted decryption failure attacks [8]. Further investigation into the security of this approach is necessary, and this will be the focus of our future work.

Acknowledgements The authors appreciate the anonymous reviewers for their valuable feedback, and would like to especially thank our shepherd, Prof. Ruben Niederhagen, for his detailed and constructive guidance during the revision process. This work is supported by the National Natural Science Foundation of China under Grant U2336210, 62172395; and Beijing Natural Science Foundation under Grant L234085.

References

1. Bar-El, H., Choukri, H., Naccache, D., Tunstall, M., Whelan, C.: The sorcerer's apprentice guide to fault attacks. Cryptology ePrint Archive, Report 2004/100 (2004), https://eprint.iacr.org/2004/100
2. Bettale, L., Montoya, S., Renault, G.: Safe-error analysis of post-quantum cryptography mechanisms. Cryptology ePrint Archive, Report 2021/1339 (2021), https://eprint.iacr.org/2021/1339
3. Biham, E., Shamir, A.: Differential fault analysis of secret key cryptosystems. In: Kaliski Jr., B.S. (ed.) CRYPTO'97. LNCS, vol. 1294, pp. 513–525. Springer, Berlin, Heidelberg (Aug 1997). https://doi.org/10.1007/BFb0052259
4. Bos, J., Ducas, L., Kiltz, E., Lepoint, T., Lyubashevsky, V., Schanck, J.M., Schwabe, P., Stehlé, D.: CRYSTALS – Kyber: a CCA-secure module-lattice-based KEM. Cryptology ePrint Archive, Report 2017/634 (2017), https://eprint.iacr.org/2017/634
5. Bos, J.W., Gourjon, M., Renes, J., Schneider, T., van Vredendaal, C.: Masking Kyber: First- and higher-order implementations. IACR TCHES **2021**(4), 173–214 (2021). https://doi.org/10.46586/tches.v2021.i4.173-214, https://tches.iacr.org/index.php/TCHES/article/view/9064
6. Coron, J.S., Gérard, F., Montoya, S., Zeitoun, R.: High-order table-based conversion algorithms and masking lattice-based encryption. IACR TCHES **2022**(2), 1–40 (2022). https://doi.org/10.46586/tches.v2022.i2.1-40
7. Dachman-Soled, D., Ducas, L., Gong, H., Rossi, M.: LWE with side information: Attacks and concrete security estimation. In: Micciancio, D., Ristenpart, T. (eds.) CRYPTO 2020, Part II. LNCS, vol. 12171, pp. 329–358. Springer, Cham (Aug 2020). https://doi.org/10.1007/978-3-030-56880-1_12
8. D'Anvers, J.P., Guo, Q., Johansson, T., Nilsson, A., Vercauteren, F., Verbauwhede, I.: Decryption failure attacks on IND-CCA secure lattice-based schemes. In: Lin, D., Sako, K. (eds.) PKC 2019, Part II. LNCS, vol. 11443, pp. 565–598. Springer, Cham (Apr 2019). https://doi.org/10.1007/978-3-030-17259-6_19
9. Delvaux, J.: Roulette: A diverse family of feasible fault attacks on masked Kyber. IACR TCHES **2022**(4), 637–660 (2022). https://doi.org/10.46586/tches.v2022.i4.637-660
10. Dobraunig, C., Eichlseder, M., Korak, T., Mangard, S., Mendel, F., Primas, R.: SIFA: Exploiting ineffective fault inductions on symmetric cryptography. IACR TCHES **2018**(3), 547–572 (2018). https://doi.org/10.13154/tches.v2018.i3.547-572, https://tches.iacr.org/index.php/TCHES/article/view/7286
11. Fahr, M., Kippen, H., Kwong, A., Dang, T., Lichtinger, J., Dachman-Soled, D., Genkin, D., Nelson, A., Perlner, R.A., Yerukhimovich, A., Apon, D.: When frodo flips: End-to-end key recovery on FrodoKEM via rowhammer. In: Yin, H., Stavrou, A., Cremers, C., Shi, E. (eds.) ACM CCS 2022. pp. 979–993. ACM Press (Nov 2022). https://doi.org/10.1145/3548606.3560673
12. Fujisaki, E., Okamoto, T.: Secure integration of asymmetric and symmetric encryption schemes. In: Wiener, M.J. (ed.) CRYPTO'99. LNCS, vol. 1666, pp. 537–554. Springer, Berlin, Heidelberg (Aug 1999). https://doi.org/10.1007/3-540-48405-1_34
13. Hermelink, J., Mårtensson, E., Samardjiska, S., Pessl, P., Rodosek, G.D.: Belief propagation meets lattice reduction: Security estimates for error-tolerant key recovery from decryption errors. IACR TCHES **2023**(4), 287–317 (2023). https://doi.org/10.46586/tches.v2023.i4.287-317

14. Hermelink, J., Pessl, P., Pöppelmann, T.: Fault-enabled chosen-ciphertext attacks on kyber. In: Adhikari, A., Küsters, R., Preneel, B. (eds.) INDOCRYPT 2021. LNCS, vol. 13143, pp. 311–334. Springer, Cham (Dec 2021). https://doi.org/10.1007/978-3-030-92518-5_15
15. Kundu, S., Chowdhury, S., Saha, S., Karmakar, A., Mukhopadhyay, D., Verbauwhede, I.: Carry your fault: A fault propagation attack on side-channel protected LWE-based KEM. IACR TCHES **2024**(2), 844–869 (2024). https://doi.org/10.46586/tches.v2024.i2.844-869
16. Mishra, N., Mukhopadhyay, D.: Probabilistic algorithms with applications to countering fault attacks on lattice based post-quantum cryptography. Cryptology ePrint Archive, Paper 2024/551 (2024), https://eprint.iacr.org/2024/551, https://eprint.iacr.org/2024/551
17. Mondal, P., Kundu, S., Bhattacharya, S., Karmakar, A., Verbauwhede, I.: A practical key-recovery attack on LWE-based key-encapsulation mechanism schemes using rowhammer. In: Pöpper, C., Batina, L. (eds.) ACNS 24International Conference on Applied Cryptography and Network Security, Part III. LNCS, vol. 14585, pp. 271–300. Springer, Cham (Mar 2024). https://doi.org/10.1007/978-3-031-54776-8_11
18. Oder, T., Schneider, T., Pöppelmann, T., Güneysu, T.: Practical CCA2-secure masked Ring-LWE implementations. IACR TCHES **2018**(1), 142–174 (2018). https://doi.org/10.13154/tches.v2018.i1.142-174, https://tches.iacr.org/index.php/TCHES/article/view/836
19. O'Flynn, C., Chen, Z.D.: ChipWhisperer: An open-source platform for hardware embedded security research. In: Prouff, E. (ed.) COSADE 2014. LNCS, vol. 8622, pp. 243–260. Springer, Cham (Apr 2014). https://doi.org/10.1007/978-3-319-10175-0_17
20. Pessl, P., Prokop, L.: Fault attacks on CCA-secure lattice KEMs. IACR TCHES **2021**(2), 37–60 (2021). https://doi.org/10.46586/tches.v2021.i2.37-60, https://tches.iacr.org/index.php/TCHES/article/view/8787
21. Ravi, P., Chattopadhyay, A., Baksi, A.: Side-channel and fault-injection attacks over lattice-based post-quantum schemes (Kyber, Dilithium): Survey and new results. Cryptology ePrint Archive, Report 2022/737 (2022), https://eprint.iacr.org/2022/737
22. 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.) COSADE 2019. LNCS, vol. 11421, pp. 232–250. Springer, Cham (Apr 2019). https://doi.org/10.1007/978-3-030-16350-1_13
23. Schwabe, P., Avanzi, R., Bos, J., Ducas, L., Kiltz, E., Lepoint, T., Lyubashevsky, V., Schanck, J.M., Seiler, G., Stehlé, D., Ding, J.: CRYSTALS-KYBER. Tech. rep., National Institute of Standards and Technology (2022), available at https://csrc.nist.gov/Projects/post-quantum-cryptography/selected-algorithms-2022
24. 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.) ASIACRYPT 2021, Part II. LNCS, vol. 13091, pp. 33–61. Springer, Cham (Dec 2021). https://doi.org/10.1007/978-3-030-92075-3_2
25. Yen, S.M., Joye, M.: Checking before output may not be enough against fault-based cryptanalysis. IEEE Transactions on Computers **49**(9), 967–970 (2000). https://doi.org/10.1109/12.869328

A Kyber

Kyber consists of three parts: key generation (Algorithm 3), encapsulation (Algorithm 4), and decapsulation (Algorithm 5) [23]. The KyberKEM is equipped with three parameter sets, corresponding to different security levels, which are defined by five individual parameters: k, η_1, η_2, d_u and d_v. Additionally, two constants n and q are used throughout the algorithm. The three parameter sets are given in Table 8. Also, Table 8 illustrates the negligible decapsulation failure probability inherent in Kyber.

Table 8. Parameter sets of Kyber.

	n	q	k	η_1	η_2	d_u	d_v	Decapsulation failure rate
Kyber512	256	3329	2	3	2	10	4	$2^{-138.8}$
Kyber768	256	3329	3	2	2	10	4	$2^{-164.8}$
Kyber1024	256	3329	4	2	2	11	5	$2^{-174.8}$

The decapsulation of Kyber comprises a decryption procedure (Algorithm 7), an encryption procedure (Algorithm 8) and the hash function-based symmetric primitives G, J. During the decapsulation process, the initial step is to perform decryption. The ciphertext is decrypted to an message $m \in \mathbb{B}^{32}$ using the secret key $\mathbf{s}$, whose coefficients belong to $[-\eta_1, \eta_1]$ and follow a centered binomial distribution. The message is subsequently re-encrypted to prevent outside exposure. The re-encrypted ciphertext is compared with the original ciphertext in the final step of the decapsulation process. If the two ciphertexts do not match, indicating a decryption error or tampering, the decapsulation procedure returns an error shared key derived from the original ciphertext.

Additionally, to reduce the ciphertext size, Kyber applies lossy compression to the coefficients in $\mathbf{u}$ and v, compressing each coefficient to d_u or d_v bits, respectively. The compression and decompression procedure can be found in Equation 9 and Equation 10. When $d = 1$, the Compress and Decompress functions are commonly referred to as message decoding and encoding in the general context of lattice-based schemes. Henceforth in this paper, the terms "encoding" and "decoding" will be utilized.

$$\text{Compress}_q(z, d) = \lceil (2^d/q) * z \rfloor \mod 2^d \tag{9}$$

$$\text{Decompress}_q(z, d) = \lceil (q/2^d) * z \rfloor \tag{10}$$

Algorithm 3 KyberKEM.KeyGen

```
Output: $pk \in \mathbb{B}^{384k+32}$
Output: $sk \in \mathbb{B}^{768k+96}$
 1: $z \xleftarrow{\$} \mathbb{B}^{32}$
 2: $(pk', sk') \leftarrow$ KyberPKE.KeyGen()
 3: $pk \leftarrow pk'$
 4: $sk \leftarrow (pk||sk'||\mathsf{H}(pk)||z)$
 5: return $(pk, sk)$
```

Algorithm 4 KyberKEM.Encaps

```
Input: $pk \in \mathbb{B}^{384k+32}$
Output: shared key $K \in \mathbb{B}^{32}$
Output: ciphertext $c \in \mathbb{B}^{32d_uk+d_v}$
 1: $m \xleftarrow{\$} \mathbb{B}^{32}$
 2: $(K, r) \leftarrow G(m||H(pk))$
 3: $c \leftarrow$ KyberPKE.Enc(pk, m, r)
 4: return $(K, c)$
```

Algorithm 5 KyberKEM.Decaps

```
Input: ciphertext $c \in \mathbb{B}^{32d_uk+d_v}$
Input: $sk \in \mathbb{B}^{768k+96}$
Output: shared key $K \in \mathbb{B}^{32}$
 1: $sk' \leftarrow sk[0 : 384k]$
 2: $pk' \leftarrow sk[384k : 768k + 32]$
 3: $h \leftarrow sk[768k + 32 : 768k + 64]$
 4: $z \leftarrow sk[768k + 64 : 768k + 96]$
 5: $m' \leftarrow$ KyberPKE.Dec$(sk', c)$
 6: $(K', r') \leftarrow G(m', h)$
 7: $\bar{K} \leftarrow J(z||c, 32)$
 8: $c' \leftarrow$ KyberPKE.Enc$(pk', m', r')$
 9:
10: if $c \neq c'$ then
11:     $K' \leftarrow \bar{K}$
12: end if
13: return $K'$
```

Algorithm 6 KyberPKE.KeyGen

```
Output: $pk' \in \mathbb{B}^{384k+32}$
Output: $sk' \in \mathbb{B}^{384k}$
 1: $d \xleftarrow{\$} \mathbb{B}^{32}$
 2: $(\rho, \delta) \leftarrow G(d)$
 3: $\mathbf{A} \leftarrow \mathcal{U}(\rho)$
 4: $(\mathbf{s}, \mathbf{e}) \leftarrow \mathcal{X}(\delta)$
 5: $\mathbf{t} \leftarrow \mathbf{A} \circ \mathbf{s} + \mathbf{e}$
 6: $pk' \leftarrow \text{Pack}(\mathbf{t}||\rho)$
 7: $sk' \leftarrow \text{Pack}(\mathbf{s})$
 8: return $(pk', sk')$
```

Algorithm 7 KyberPKE.Dec

```
Input: $sk' \in \mathbb{B}^{384k}$
Input: ciphertext $c \in \mathbb{B}^{32(d_uk+d_v)}$
Output: message $m \in \mathbb{B}^{32}$
 1: $c_1 \leftarrow c[0 : 32d_uk]$
 2: $c_2 \leftarrow c[32d_uk : 32(d_uk + dv)]$
 3: $\mathbf{u}_l \leftarrow \text{Decompress}_{d_u}(\text{Unpack}(c_1))$
 4: $v_l \leftarrow \text{Decompress}_{d_v}(\text{Unpack}(c_2))$
 5: $\hat{\mathbf{s}} \leftarrow \text{Unpack}_{12}(sk')$
 6: $mp \leftarrow v_l - \mathbf{s} \circ \mathbf{u}_l$
 7: $m \leftarrow \text{Pack}_1(\text{Compress}_1(mp))$
 8: return $m$
```

Algorithm 8 KyberPKE.Enc

```
Input: $pk' \in \mathbb{B}^{384k+32}$
Input: message $m \in \mathbb{B}^{32}$
Input: randomness $r \in \mathbb{B}^{32}$
Output: ciphertext $c \in \mathbb{B}^{32(d_uk+d_v)}$
 1: $\mathbf{t} \leftarrow \text{Unpack}(pk'[0 : 384k])$
 2: $\rho \leftarrow pk'[384k : 384k + 32]$
 3: $\mathbf{A} \leftarrow \mathcal{U}(\rho)$
 4: $\mathbf{r}, \mathbf{e}_1, e_2 \leftarrow \mathcal{X}(r)$
 5: $\mathbf{u} = \mathbf{A} \circ \mathbf{r} + \mathbf{e}_1$
 6: $\mu \leftarrow \text{Decompress}_1(\text{Unpack}_1(m))$
 7: $v \leftarrow \mathbf{t} \circ \mathbf{r} + e_2 + \mu$
 8: $c_1 \leftarrow \text{Pack}_{d_u}(\text{Compress}_{d_u}(\mathbf{u}))$
 9: $c_2 \leftarrow \text{Pack}_{d_v}(\text{Compress}_{d_v}(v))$
10: return $c \leftarrow (c_1||c_2)$
```

B Masked Decoder

The original masked decoder presented in [5] is shown in Equation 11. During our analysis, we identified a potential issue at the boundary condition $x = \lfloor \frac{q}{4} \rfloor$, which can cause an error in the decryption result. The authors have clarified this issue, and the corrected version of the function, which resolves the edge case, is provided in Equation 1.

$$\text{Compress}_q^s(z) = z_{11} \oplus (\neg z_{11} \cdot z_{10} \cdot z_9 \cdot (z_8 \oplus (\neg z_8 \cdot z_7))) \tag{11}$$