IACR Transactions on Cryptographic Hardware and Embedded Systems
ISSN 2569-2925, Vol. 2024, No. 2, pp. 844–869. DOI:10.46586/tches.v2024.i2.844-869

Carry Your Fault: A Fault Propagation Attack on Side-Channel Protected LWE-based KEM

Suparna Kundu[1], Siddhartha Chowdhury[2], Sayandeep Saha[3], Angshuman Karmakar[1,4], Debdeep Mukhopadhyay[2] and Ingrid Verbauwhede[1]

[1] COSIC, KU Leuven, Leuven, Belgium
[2] Indian Institute of Technology Kharagpur, Kharagpur, India
[3] Université catholique de Louvain, Ottignies-Louvain-la-Neuve, Belgium
[4] Indian Institute of Technology Kanpur, Kanpur, India
{suparna.kundu,angshuman.karmakar,ingrid.verbauwhede}@esat.kuleuven.be
{siddhartha.chowdhury92,sayandeep.iitkgp,debdeep.mukhopadhyay}@gmail.com

Abstract.
Post-quantum cryptographic (PQC) algorithms, especially those based on the learning with errors (LWE) problem, have been subjected to several physical attacks in the recent past. Although the attacks broadly belong to two classes – passive side-channel attacks and active fault attacks, the attack strategies vary significantly due to the inherent complexities of such algorithms. Exploring further attack surfaces is, therefore, an important step for eventually securing the deployment of these algorithms. Also, it is important to test the robustness of the already proposed countermeasures in this regard. In this work, we propose a new fault attack on side-channel secure masked implementation of LWE-based key-encapsulation mechanisms (KEMs) exploiting fault propagation. The attack typically originates due to an algorithmic modification widely used to enable masking, namely the Arithmetic-to-Boolean (`A2B`) conversion. We exploit the data dependency of the adder carry chain in `A2B` and extract sensitive information, albeit masking (of arbitrary order) being present. As a practical demonstration of the exploitability of this information leakage, we show key recovery attacks of Kyber, although the leakage also exists for other schemes like Saber. The attack on Kyber targets the decapsulation module and utilizes Belief Propagation (BP) for key recovery. To the best of our knowledge, it is the first attack exploiting an algorithmic component introduced to ease masking rather than only exploiting the randomness introduced by masking to obtain desired faults (as done by Delvaux [Del22]). Finally, we performed both simulated and electromagnetic (EM) fault-based practical validation of the attack for an open-source first-order secure Kyber implementation running on an STM32 platform.

Keywords: Post-quantum cryptography · Fault attack · Key-encapsulation mechanism · Masked implementation · arithmetic to Boolean conversion

1 Introduction

Post-quantum cryptography (PQC) comprises cryptosystems that are designed using hard problems that an adversary cannot solve *easily* using a sizeable quantum computer. PQC has been developed as a contingency plan in anticipation of the invention of sizeable quantum computers that can break our prevalent classical cryptosystems in the near future. A remarkable step in this direction is the recently concluded post-quantum standardization procedure initiated by the National Institute of Standards and Technology (NIST). NIST proposed some standard Digital Signatures, Key-Encapsulation Mechanisms (KEM), and Public-Key Encryption schemes [AAC+22], which can be used to replace our current cryptosystems.

Licensed under Creative Commons License CC-BY 4.0.
Received: 2023-10-15 Accepted: 2023-12-15 Published: 2024-03-12

Therefore, we should put more focus on practical issues such as the physical security of PQC so that it generates enough confidence to be deployed in the near future.

Countermeasures for two different types of physical attacks *i.e.* side-channel attacks (SCA) and Fault Attacks (FA) differ greatly from each other. Usually, countermeasures for one type of physical attack do not guarantee any security against another type of physical attack. However, if an SCA countermeasure is not applied carefully, its application sometimes opens up new vulnerable points for SCA. One such attack is shown in [FRVD08], where the authors exploited carry leakage from the randomized exponent of RSA [RSA78]. Masking [CJRR99] is a well-known and provably secure countermeasure against SCA. Masking alone provides no assurance of security against FAs, except in some special cases of Statistical Ineffective Fault Attack (SIFA) [SJR+19][1]. However, protection against both SCA and FA is desired. Therefore, it is important to understand the robustness of each dedicated countermeasure against cross-attacks, such as FA on SCA-secure implementation or vice-versa. It is especially important to understand whether one such countermeasure results in some new vulnerabilities that are exploitable by attack vectors. Recent work on symmetric key ciphers has shown that the interaction between two countermeasure classes is tricky and often leads to new attacks [SBJ+21, SRJB23]. We find that there is a lack of study on such *cross-attack* strategies *i.e.* FAs on schemes with SCA countermeasures or vice versa for PQC schemes. Therefore, in this work, we ask: *Does one countermeasure create new vulnerabilities for another class of attacks in the PQC context?* Specifically, since there exist many SCA secure masking schemes [BDK+20, BGR+21, HKL+22, KDVB+22] for learning with errors (LWE) [Reg09]-based PQC key-encapsulation mechanisms (KEM), in this work, we will limit our focus to these schemes only.

Most of the LWE-based KEMs (e.g. Kyber [BDK+17], Saber [DKRV18], NewHope [ADPS16], etc.) use a variant of the Fujisaki-Okamoto (FO) [FO99] transformation [JZC+17] to achieve security against Chosen Ciphertext Attack (CCA) attacks. A CCA-secure KEM consists of three algorithms: Key generation, Encapsulation, and Decapsulation. Although the key generation and encapsulation algorithms can be attacked using fault or side-channel [RRB+19, VOGR18], the decapsulation algorithm is especially exploitable as it contains the secret key [PP19]. The secret key of KEM is non-ephemeral *i.e.* the same secret key is repeatedly used in the decapsulation for different ciphertexts from various sources. Therefore, an attacker can refine their attack and improve their success probability by observing multiple traces with different inputs or by applying the fault in different executions while the key remains fixed. The decapsulation operation includes decryption and re-encryption and only returns a session key if the given ciphertext and re-encrypted ciphertext are the same. Otherwise, it returns a random key indicating decryption failure. These steps of the FO transformation inherently resist DFA [OSPG18, TMA11], which require faulty outcomes to proceed. However, a fault to bypass the equality check (between the received ciphertext and the re-encrypted ciphertext) can enable DFA. While this attack is simple, it is not the only attack possible. More precisely, equality check is not the only place where faults can be injected. Generally, ineffective FAs, similar to SIFA [SJR+19] and Fault Template Attacks (FTA) [SBBR+20], have been quite successful so far [PP21, HPP21, Del22]. Such attacks extract information from two events: a) in case there is decryption failure due to faults (effective fault), and b) the decryption succeeds even though a fault has been injected (ineffective fault). The "effectiveness" of a fault depends on the decryption noise, which depends on the long-term secret key $\boldsymbol{s}$. Therefore, each decryption failure or success provides an inequality involving $\boldsymbol{s}$. The attacker can accumulate a sufficient number of such inequalities by injecting several faults and can retrieve $\boldsymbol{s}$ by solving this system of inequalities.

Several works [BMR21, PP21, HPP21, Del22] have proposed efficient ineffective FAs on LWE-based KEM schemes. In [PP21], Pessl et al. used an instruction-skip fault in the

[1] Masking alone provides some SIFA security only for cases where faults cannot be injected inside S-Boxes and the faulty outputs are blocked leaving SIFA as the only option (ruling out Differential Fault Attacks aka. DFAs). However, this guarantee does not hold when faults can corrupt the S-Box internals and fine-grained error correction is needed along with masking.

`Decode` part (Figure 3) of the decapsulation of Kyber and then observed if this fault resulted in the correct session key or not. The attack proposed in [HPP21] is similar to [PP21] but uses a different location to inject fault, which also belongs to the category of ineffective fault attack. Later, Jeroen Delvaux extended the fault locations of [HPP21] and improved the inequality solver in [Del22]. There is another recent class of attack (not ineffective faults), which injects faults in the constants of the Number-Theoretic-Transform (NTT) computation, reducing the entropy of the LWE instances leading to key recovery [RYB+23]. Notably, in all of these cases, the main difference comes from the fault location – finding out new attack location is always beneficial from a security assessment perspective. It also indicates where one should put the countermeasures and what kind of countermeasures are needed. In this regard, the contributions of this paper are as follows:

- We propose an FA by injecting faults in a previously unexplored location present in many LWE-based KEMs. The attack falls in the category of ineffective FAs. However, the new location we explore (the Arithmetic-to Boolean mask conversion module aka. `A2B`) is an artifact of masking. Therefore, our attack can be seen as a "side-effect" of masking. There exists a previous attack due to [Del22] which is aided by masking. The reason was that masking randomizes the variable to be faulted and increases the chance of realizing a desired (e.g. single-bit) fault model with different executions. Our exploration is different – we show that an algorithmic change, which is essential for efficient masking, is problematic. Also, we show later in this paper that preventing these two attacks has very different requirements, and we anticipate that our attack would be difficult to prevent using standard techniques, such as duplication (in Sec. 6).

- The proposed vulnerability originates from a fault-induced information-leakage channel through the carry propagation logic of a masked adder. While we restrict our discussion mostly to Kyber in this paper, the channel exists for Saber (we will explain this briefly) and maybe for many other schemes that utilize such adders. Moreover, the leakage is oblivious to the masking order, and, therefore, the attack remains unaffected by the increase in SCA security.

- We realize the key recovery algorithm using the Belief Propagation algorithm provided in [Del22]; however, with important customizations needed for our purpose. While the exact version of the attack requires a single-bit fault at a specific bit, we show that the effectiveness remains unchanged for much relaxed multi-bit fault models – although at the cost of more observations. This interesting observation is due to the nature of carry propagation through the adder and the restricted range and distribution of values the message coefficients can take.

- Our final contribution is a practical realization of the attack on an STM32-based open-source first-order masked implementation of Kyber. We used electromagnetic (EM) faults for this purpose and effectively recovered the key with 1.9 million injections.

It is worth mentioning that there exist multiple works targeting the masked implementations of KEMs using SCA analysis [NDGJ21, NWDP22]. Most of them, such as [NDGJ21], exploit the inherent weaknesses of masked software implementations of these algorithms due to low noise and microarchitectural leakages. The attack proposed by us depends on the algorithm and, therefore, does not depend much on such implementation-specific physical assumptions other than that of the fault. In other words, our attack remains unchanged for hardware implementations, provided one can realize the desired fault model, even if the masking is implemented very carefully with sufficient noise.

2 Preliminaries

Notations: The k-th bit of an integer $u \in \mathbb{Z}$ is denoted as $u^{(k)}$. $\mathbb{Z}_q$ is a ring of integer modulo q and R_q is a polynomial ring $\mathbb{Z}_q[X]/(X^n+1)$. R_q^l is a ring containing vectors with l polynomials of R_q and we use $R_q^{l \times l}$ to represent a ring with $l \times l$ matrices over R_q. We denote a single polynomial with lower case letter (e.g., $x \in R_q$), a vector of polynomials with bold lower case letter (e.g., $\boldsymbol{y} \in R_q^l$), and a matrix of polynomials with bold upper case letter (e.g., $\boldsymbol{A} \in R_q^{l \times l}$). We denote the i-th ($i \in \{0, 1, ..., n-1\}$) coefficient of a polynomial x as $x[i]$ and the i-th ($i \in \{0, 1, ..., n-1\}$) coefficient of j-th ($j \in \{0, 1, ..., l-1\}$) polynomial in a vector of polynomials $\mathbf{y}$ as $\mathbf{y}[j][i]$. If an element v is sampled from a set S according to a distribution χ, we present it by $v \leftarrow \chi(S)$. However, if the element v is generated from a $seed_v$ with the help of a pseudorandom number generator and follows the distribution χ over the set S, then we denote it by $v \leftarrow \chi(S; seed_v)$. We use $\mathcal{U}$ as uniform distribution, and β_μ as Centered Binomial Distribution (CBD) with standard deviation $\sqrt{\mu/2}$. $\oplus$ denotes bit-wise XOR operation and & denotes bit-wise AND operation. $\lfloor w \rceil$ represents the rounding operation that outputs the closest integer of w and is rounded upwards in case of ties (i.e., if $w = 1/2$ then $\lfloor w \rceil = 1$). $w \ll b$ and $w \gg b$ denotes logical shifting of w by b positions to left or right respectively. Here, w is an integer, and b is a natural number. These operations can be extended for a polynomial x by performing them coefficient-wise (e.g., $x \gg b = x[i] \gg b$, for each $i \in \{0, 1, ..., n-1\}$). If $u \in \{0,1\}$ then $\overline{u} = 1 \oplus u$.

2.1 LPR Public-Key Encryption

The Ring-LWE (RLWE) [LPR10] problem is a variant of the Regev's LWE [Reg09] problem introduced by Lyubashevsky et al. The decision version of RLWE problem states that if $a \leftarrow \mathcal{U}(R_q)$, $s, e \leftarrow \chi(R_q)$, $b' \leftarrow \mathcal{U}(R_q)$, and $b = as + e$, then distinguishing (a, b) and (a, b') is hard. The authors also proposed a public-key encryption scheme based on RLWE problem, which is commonly known as the LPR scheme. This scheme consists of three algorithms: (i) key generation (`LPR.PKE.KeyGen`), (ii) encryption (`LPR.PKE.Enc`), and (iii) decryption (`LPR.PKE.Dec`), which are presented in Figure 1. The key generation algorithm generates the public key and secret key pairs. In this algorithm, secret polynomial s and the noise polynomial e are sampled using a narrow distribution χ over the set R_q. The public polynomial a's coefficients are drawn using a uniform distribution over the set $\mathbb{Z}_q$, and the other part of the public key is the polynomial b is computed by computing $as + e$. The Encryption algorithm takes input as the public key pk and message m and produces the ciphertext pair (u, v). Here, u is the key containing part of the ciphertext and generated like b in key generation. v is the message containing components of the ciphertext. To generate v, an n-bit message m is encoded to a message polynomial with mapping `Encode`. One such mapping can be `Encode`$: R_2 \rightarrow R_q$ defined by $\texttt{Encode}(m) = \sum_{i=1}^{n} \lfloor q/2 \rceil m[i]$, and then added with $bs' + e_2$. The decryption algorithm computes $m' = v - us$ to obtain a noisy version of the message m.

$$\begin{aligned} m' &= v - us = bs' + e_2 + \texttt{Encode}(m) - (as' + e_1)s \\ &= (as+e)s' + e_2 + \texttt{Encode}(m) - (as' + e_1)s = \texttt{Encode}(m) + es' - e_1 s + e_2 \end{aligned}$$

Here, $es' - e_1 s + e_2$ is known as decryption noise. Also, e, s', e_1, s, and e_2 are all sampled from a small distribution χ. So, the decryption noise can be removed with a very high probability. This is performed by the `Decode` operation, which takes input as $m' \in R_q$ and outputs $m \in R_2$ (shown in Figure 3).

The chosen plaintext attack (CPA)-secure `LPR.PKE` can be converted to a CCA (chosen ciphertext attack)-secure key encapsulation mechanism (KEM) using a post-quantum variant of the Fujisaki-Okamoto (FO) transformation proposed by Jiang et al. [JZC+17].

We call this FO transformation integrated CCA secure KEM as `LPR.KEM`, and present in Figure 2. $\mathcal{G}$, $\mathcal{H}$, and `KDF` are three hash functions employed in this KEM as a part of FO transformation. This FO transformation is used in Kyber and Saber.

```
LPR.PKE.KeyGen()
1. a ← U(R_q)
2. s, e ← χ(R_q)
3. b = (as + e) ∈ R_q^l
4. return (pk = (a, b), sk = (a, s))

LPR.PKE.Enc(pk = (a, b), message m ∈ Z_2^n)
1. s', e_1, e_2 ← χ(R_q)
2. u = (as' + e_1) ∈ R_q
3. v = bs' + e_2 + Encode(m)
4. return c = (u, v)

LPR.PKE.Dec(sk = (a, s), c = (u, v))
1. m' = v − us ∈ R_q
2. m = Decode(m') ∈ R_2
3. return m
```

Figure 1: LPR.PKE [LPR10]

```
LPR.KEM.KeyGen()
1. (pk, sk) = LPR.PKE.KeyGen()
2. pkh = H(pk)
3. z ← U({0, 1}^n)
4. return (pk, s̃k = (sk, z, pkh))

LPR.KEM.Encaps(pk)
1. m ← U({0, 1}^n)
2. m = H(m)
3. (K̂, r) = G(H(pk), m)
4. c = LPR.PKE.Enc(pk, m; r)
5. K = KDF(K̂, H(c))
6. return (c, K)

LPR.KEM.Decaps(s̃k = (sk, z, pkh), pk, c)
1. m = LPR.PKE.Dec(sk, c)
2. (K̂', r') = G(pkh, m)
3. c_* = LPR.PKE.Enc(pk, m; r')
4. if: c = c_*
5.    return K = KDF(K̂', H(c))
6. else:
7.    return K = KDF(z, H(c))
```

Figure 2: CCA secure LPR.KEM [JZC+17]

2.2 PQ KEM Schemes Kyber and Saber

Kyber is a LPR.KEM like CCA secure KEM, but its security is based on the Module Learning with Errors (MLWE) problem. Here, the public polynomial of LPR.PKE a is a matrix of polynomials $\boldsymbol{A} \in R_q^{l\times l}$, the secret polynomial s is a vector of polynomials $\boldsymbol{s} \in R_q^l$, and the error polynomial e is a vector of polynomials $\boldsymbol{e} \in R_q^l$. The MLWE problem states that if $\boldsymbol{A} \leftarrow \mathcal{U}(R_q^{l\times l})$, $\boldsymbol{s} \leftarrow \beta_\mu(R_q^l)$, $\boldsymbol{e} \leftarrow \beta_\mu(R_q^l)$, $\boldsymbol{b'} \leftarrow \mathcal{U}(R_q^l)$, and $\boldsymbol{b} = (\boldsymbol{As}) + \boldsymbol{e}$, then $(\boldsymbol{A}, \boldsymbol{b})$ and $(\boldsymbol{A}, \boldsymbol{b'})$ are hard to distinguish. Kyber uses number-theoretic transformations to perform polynomial multiplication. Three security versions of Kyber named Kyber512, Kyber768, and Kyber1024, depending on different values of the parameter set, and are presented in Table 1. The parameter set of Kyber contains prime modulus q and two power-of-two moduli p and t, which form the rings R_q, R_p, R_t. In the Kyber.PKE.Enc, the Compress function shortens each coefficient of the ciphertext $c = (u, v)$. Each coefficient of u is reduced from R_q to R_p, and each coefficient of v is reduced from R_q to R_t. Conversely, the Decompress function in Kyber.PKE.Dec, extend the ciphertext bits and maps each element of R_p or R_t to R_q. The $\texttt{Encode}: \{0,1\}^n \to R_q$ function converts n-bit message to a polynomial in R_q, and the $\texttt{Decode}: R_q \to \{0,1\}^n$ function reverse the effect of the Encode function.This is shown in Figure 3. Kyber uses two CBD distributions β_{η_1} and β_{η_2} to sample secret and error vector. Here, we brief the Kyber scheme and recommend [BDK+17] for additional details.

Saber is also a CCA secure KEM, and its security depends on the Module Learning with Rounding (MLWR) problem. The MLWR problem states that if $\boldsymbol{A} \leftarrow \mathcal{U}(R_q^{l\times l})$, $\boldsymbol{s} \leftarrow \beta_\mu(R_q^l)$,

`Compress`(v', t) (referred as `Comp`)	`Decompress`(v, t) (referred as `Decomp`)
1. $v = \lfloor (t/q)v' \rceil \bmod t$	1. $v' = \lfloor (q/t)v \rceil$
2. **return** (v)	2. **return** (v')
`Encode`(m)	`Decode`(m')
1. for `i=1 to n`:	1. for `i=1 to n`:
2. $m'[i] = \lfloor q/2 \rceil m[i] \in \mathbb{Z}_q$	2. $m[i] = (((m'[i] \ll 1) + \lfloor q/2 \rceil)/q) \& 1 \in \mathbb{Z}_2$
3. **return** $(m' \in R_q)$	3. **return** $(m \in \{0,1\}^n)$

Figure 3: `Compress`, `Decompress`, `Encode`, and `Decode` functions of `Kyber` [BDK+17]

Table 1: Parameters of Kyber with security and failure probability [BDK+17]

Scheme Name	Parameters							Post-quantum Security	Failure Probability	NIST Security Level
	l	n	q	p	t	η_1	η_2			
Kyber512	2	256	3329	2^{10}	2^4	3	2	2^{107}	2^{-139}	1
Kyber768	3			2^{10}	2^4	2	2	2^{165}	2^{-164}	3
Kyber1024	4			2^{11}	2^5	2	2	2^{232}	2^{-174}	5

$\boldsymbol{b}' \leftarrow \mathcal{U}(R_p^l)$, $q > p$ and $\boldsymbol{b} = \lfloor (q/p)(\boldsymbol{As}) \rceil$, then $(\boldsymbol{A}, \boldsymbol{b})$ and $(\boldsymbol{A}, \boldsymbol{b}')$ are hard to distinguish. Here the error vector of the MLWE problem is replaced by the rounding operation. The moduli q, p, t that are used to form the rings R_q, R_p, R_t in Saber algorithms are all power-of-two. The number of bits in each coefficient of a polynomial of the corresponding ring can be calculated as $\log_2(q) = \epsilon_q$, $\log_2(p) = \epsilon_p$, and $\log_2(t) = \epsilon_t$. The encoding and decoding operations of Saber are similar to `LPR.PKE`, and these operations are realized by using the shift operation due to the power-of-two moduli. There are three security versions of Saber named LightSaber, Saber, and FireSaber, depending on different values of the parameter set, which is shown in Table 2. We suggest the original paper [DKRV18] for further specifics.

Table 2: Parameters of Saber with security and failure probability [DKRV18]

Scheme Name	Parameters						Post-quantum Security	Failure Probability	NIST Security Level
	l	n	q	p	t	μ			
LightSaber	2	256	2^{13}	2^{10}	2^3	5	2^{107}	2^{-120}	1
Saber	3				2^4	4	2^{172}	2^{-136}	3
FireSaber	4				2^6	3	2^{236}	2^{-165}	5

2.3 Masking Saber and Kyber

Due to many successful SCAs [PP19, KPP20, MBB+22, RRCB20] on the decapsulation algorithm of the NIST KEM candidates, many masked implementations for those schemes have been proposed to prevent SCA [BDK+20, HKL+22]. In this work, we also target the masked decapsulation procedure of an LWE-based KEM. Therefore, we briefly describe the first-order masked decapsulation algorithm below, which can prevent first-order SCA.

The decapsulation of a general LWE-based KEM follows three steps: (i) decryption, (ii) re-encryption, and (iii) ciphertext comparison (Figure 2). We must mask each of these three steps to protect the decapsulation algorithm from SCA. In first-order masking, the sensitive variable w is split into two shares, w_1 and w_2. The relation between w and (w_1, w_2) for arithmetic masking is $(w_1 + w_2) \bmod q = w$ and for Boolean masking it is $w_1 \oplus w_2 = w$. LWE-based KEMs primarily utilize linear polynomial arithmetic operations over a ring R_q, such as ring multiplication with one masked and another unmasked input, ring addition, and ring subtraction. These operations are duplicated and performed on each share independently in masked settings. For masking them, arithmetic masking techniques are used. The decapsulation also

has non-linear arithmetic over R_q. Those are shift operations on polynomials, and hash functions $\mathcal{H}$, $\mathcal{G}$, the extension function used for sampling pseudorandom numbers `XOF`, the CBD β_μ, and ciphertext equality checking operation =?. These operations are expressed in bit operations and masked with Boolean masking. Except for these operations, the masked decapsulation needs two share conversion algorithms *i.e.* arithmetic to Boolean (`A2B`) and Boolean to arithmetic (`B2A`) conversions. The `B2A` conversion occurs inside the masked CBD (β_μ).

In Saber's decapsulation operation, the ring is R_p instead of R_q. A pictorial version of this masked decapsulation operation of Saber is presented in Figure 4. In this work, we will focus on the part surrounded by the red rectangle in the figure, which includes the `A2B` conversion and shift operations. A version of `A2B` conversion used in Saber is shown in Algorithm 1.

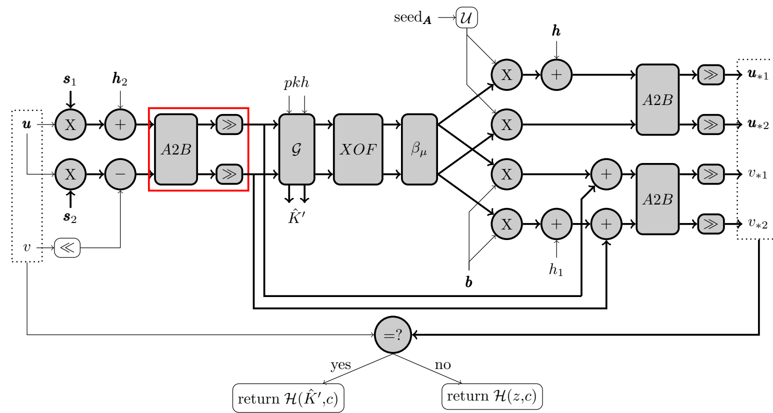


Figure 4: First-order masked `Saber.KEM.Decaps` algorithm [BDK+20]. The highlighted operations in color gray are influenced by the non-ephemeral secret-key $\boldsymbol{s}$ and use masking to prevent SCA. The component we focus on in this work is enveloped with the red rectangle.

Algorithm 1: First-order masked arithmetic to Boolean conversion (`A2B`) [CGV14]

Input : $(a_1, a_2) \in R_{2^k} \times R_{2^k}$ such that $(a_1 + a_2) \bmod 2^k = w \in R_{2^k}$
Output : $(b_1, b_2) \in R_{2^k} \times R_{2^k}$ such that $b_1 \oplus b_2 = w$

1 $u_1 \leftarrow \mathcal{U}(R_{2^k})$; $u_2 = u_1 \oplus a_1$
2 $v_1 \leftarrow \mathcal{U}(R_{2^k})$; $v_2 = v_1 \oplus a_2$
3 $(b_1, b_2) =$ `SecAdd`$((u_1, u_2), (v_1, v_2), k)$ [Algorithm 2]
4 **return**(b_1, b_2)

In Kyber, the ring modulus q is prime, the share conversion algorithms `A2B` ($\texttt{A2B}_q$) and `B2A` ($\texttt{B2A}_q$) are different compared to Saber, which uses power-of-two modulus. In Kyber, the shares are first transferred from prime to a power-of-two modulus, and the aforementioned `A2B` conversion (Algorithm 1) or a `B2A` conversion algorithm with input in power-of-two modulus is performed. For this reason, a couple of extra masked modules, such as `Decode`, `Encode`, `Compress`, and `transform-power-of-2` are required in masked Kyber. In Kyber, the `A2B` conversions are used inside the `Decode` and `Comp` components, and the `B2A` conversions are inside the `Encode`. Figure 5 is an illustrated version of the first-order masked decapsulation algorithm of Kyber. The function we exploit in this work is the `A2B` conversion in the `Decode` module, which is surrounded by the red rectangle in the figure. We present a first-order masked `Decode` procedure in Algorithm 3. According to this algorithm, (m'_{a_1}, m'_{a_2}) are arithmetically masked shares of the input of the `Decode` operation m'. We use Figure 5 to illustrate the steps of the masked decoding algorithm for a coefficient of m', say $m'[i]$. Here, the first figure shows the distribution of a coefficient of $m'[i]$. The second figure presents the distribution of that

Algorithm 2: SecAdd [CGV14]

Input : $(u_1, u_2), (v_1, v_2) \in R_{2^k} \times R_{2^k}$ such that $u_1 \oplus u_2 = \hat{u}$ and $v_1 \oplus v_2 = \hat{v}$
Output : $(z_1, z_2) \in R_{2^k} \times R_{2^k}$ such that $z_1 \oplus z_2 = (\hat{u} + \hat{v}) \bmod 2^k$

1 $(c_1^1, c_2^1) = (0, 0)$
2 $r_0, r_1, r_2 \leftarrow \mathcal{U}(R_{2^k})$
3 **for** *i=1 to k-1* **do**
4 $\quad tmp1_1 = (u_1^i \& v_1^i) \oplus r_0^i$; $tmp1_2 = (u_2^i \& v_2^i) \oplus r_0^i$
5 $\quad tmp2_1 = (u_1^i \& c_1^i) \oplus r_1^i$; $tmp2_2 = (u_2^i \& c_2^i) \oplus r_1^i$
6 $\quad tmp3_1 = (v_1^i \& c_1^i) \oplus r_2^i$; $tmp3_2 = (v_2^i \& c_2^i) \oplus r_2^i$
7 $\quad c_1^{i+1} = tmp1_1 \oplus tmp2_1 \oplus tmp3_1$; $c_2^{i+1} = tmp1_2 \oplus tmp2_2 \oplus tmp3_2$
8 $z_1 = u_1 \oplus v_1 \oplus c_1$; $z_2 = u_2 \oplus v_2 \oplus c_2$
9 **return** (z_1, z_2)

coefficient after the subtraction with $q/4$. The third figure expresses the distribution of that coefficient after the transformation from $\mathbb{Z}_q$ to $\mathbb{Z}_{2^{k+1}}$, where $k = \lceil \log_2(q) \rceil$. The last figure displays the distribution of that coefficient after the subtraction with $q/2$. After this, the `A2B` conversion takes place and produces Boolean shares $(d_{b_1}[i], d_{b_2}[i])$ in Algorithm 3. Then the most significant bits (MSBs) of $(d_{b_1}[i], d_{b_2}[i])$ are calculated by applying `MSB` coefficient-wise. $(d_{b_1}[i], d_{b_2}[i])$ act as the Boolean shares of $m[i] = (m_{b_1}[i], m_{b_2}[i])$. Here, kth and $(k+1)$th bit of $(d_{b_1}[i], d_{b_2}[i])$ both can serve as the Boolean shares of the message bit $m[i]$. Therefore,

$$(d_{b_1}[i]^{(k+1)}, d_{b_2}[i]^{(k+1)}) = (d_{b_1}[i]^{(k)}, d_{b_2}[i]^{(k)}) = (m_{b_1}[i], m_{b_2}[i]) \tag{1}$$

All other bits of $(d_{b_1}[i], d_{b_2}[i])$ act as decryption noise. As noted in Section 1, the decryption noise is linearly related to the secret key.

Algorithm 3: First-order masked decode algorithm [OSPG18]

Input : $(m'_{a_1}, m'_{a_2}) \in R_q \times R_q$ such that $m' = (m'_{a_1} + m'_{a_2}) \bmod q$, and $k = \lceil \log_2(q) \rceil$
Output : $(m_{b_1}, m_{b_2}) \in R_2 \times R_2$ where $m = m_{b_1} \oplus m_{b_2} = \texttt{Decode}((m'_{a_1} + m'_{a_2}) \bmod q)$

1 $m'_{a_1} = m'_{a_1} - \lfloor q/4 \rfloor$
`/* transfers arithmetic shares from` $\bmod q$ `to` $\bmod 2^{k+1}$ `*/`
2 $(g_{a_1}, g_{a_2}) = \texttt{transform-power-of-2}(m'_{a_1}, m'_{a_2}, (k+1))$
3 $g_{a_1} = g_{a_1} - \lfloor q/2 \rfloor$
4 $(d_{b_1}, d_{b_2}) = \texttt{A2B}((g_{a_1}, g_{a_2}), (k+1))$ [Algorithm 1]
5 $m_{b_1} = \texttt{MSB}(d_{b_1})$; $m_{b_2} = \texttt{MSB}(d_{b_2})$
6 **return** (m_{b_1}, m_{b_2})

3 Fault Propagation through the Carry Chain

In this section, we present our main observation, which results in key recovery fault attacks on the masked decapsulation algorithms of certain CCA secure LWE-based KEMs. Broadly, our attack exploits the `Decode` component (ref. Figure 6) that removes the decryption noise and produces the message in the decapsulation algorithm. However, unlike [PP21], which skips the first subtraction operation (line 1. in Algorithm 3) in the `Decode` module, we target the `A2B` algorithm. The `A2B` adds the arithmetic shares using Boolean masking and generates Boolean shares amenable to the shift operation so that the message bits can be extracted easily. More precisely, we target one of the arithmetic shares input to this module with faults. For simplicity, we limit our discussion to two shares (ref. Algorithm 3) in the next few paragraphs, and later explain the many shares case. Moreover, for ease of explanation,

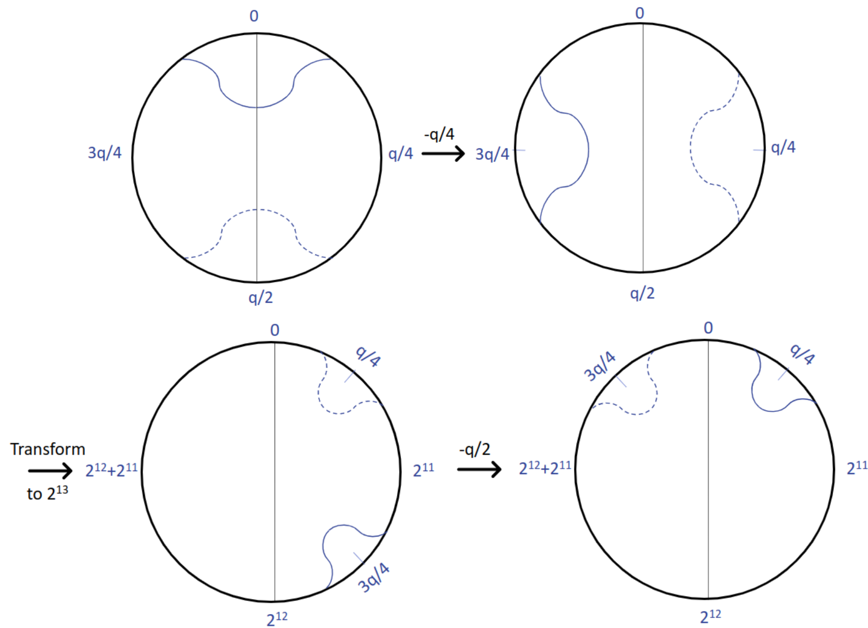


Figure 5: The steps of the masked decoding algorithm of Kyber presented in Algorithm 3[OSPG18].

we limit the discussion to the `Decode` module of Kyber only (described in the last section). The differences for the Saber case will be explained at the end.

The output of the `Decode` stage are the message bits (masked), which are inputs to the re-encryption stage. As described in the last section, these (masked) message bits are derived from the (masked) message coefficients containing (masked) noise. *The faults induced by us at the input of the* `A2B` *(at a specific bit of any one of the shares) propagate to these decoded (masked) message bits, conditioned on a specific (unmasked) bit of the noise.* In other words, whether or not the fault propagates to the (masked) message bits (and, therefore, causes a decryption failure) depends on the actual unmasked value of some noise bit. Now the noise is dependent on the secret key $\boldsymbol{s}$. Therefore, based on this fault propagation property, we can construct inequalities (each corresponding to one ciphertext) involving the secret, and solve a system of such inequalities using the Belief Propagation (BP) algorithm. The BP algorithm finally returns the long-term secret key. Note that, in this attack, we always work with valid ciphertexts, *i.e.* the ciphertexts that will be generated using proper execution of the encapsulation mechanism of the scheme. So, the probability of decryption failure (without fault) during the decapsulation procedure is identical to the preassigned decryption failure probability of the scheme, which is very small ($< 2^{-128}$). Therefore, we can safely assume that all the observed decryption failures happen due to the induced faults. Next, we explain the information leakage that we exploit due to the fault propagation through carry chains of the `A2B`.

3.1 Fault Propagation through First-order Masked `A2B`

We begin our discussion with stuck-at faults. Without loss of generality, suppose we introduce `stuck-at-1` bit-fault (on an input share) at the $(k-1)$th bit of the i-th input coefficient (*i.e.* $g_{a_1}[i]^{(k-1)}$) to the `A2B` conversion. Here $i \in \{0, 1, ..., n-1\}$[2]. Let $g = (g_{a_1} + g_{a_2}) \bmod 2^{k+1} = (d_{b_1} \oplus d_{b_2})$. Here, $g[i]^{(k+1)} = (d_{b_1}[i]^{(k+1)} \oplus d_{b_2}[i]^{(k+1)}) = (m_{b_1}[i] \oplus m_{b_2}[i]) = m[i]$ ($m[i]$s are the

[2] Please note that we start counting of the index k from 1.

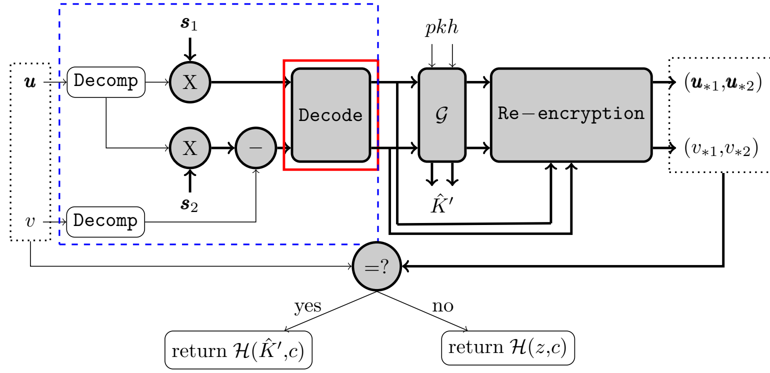


Figure 6: First-order masked decapsulation algorithm of LW(E/R) based schemes. The highlighted operations in color gray are influenced by the non-ephemeral secret-key $\boldsymbol{s} = (\boldsymbol{s}_1, \boldsymbol{s}_2)$. The function we exploit in our attack is enveloped with the red rectangle.

bits of the message). Henceforth in this section, we will only focus on a single coefficient. So, for the sake of simplicity, we rename the following:

- $g_{a_1}[i]$ as x, $g_{a_2}[i]$ as y, and $g[i]$ as z. These are $(k+1)$ bit registers.
- $m_{b_1}[i]$ as m_1, $m_{b_2}[i]$ as m_2, and $m[i]$ as $\hat{m}$. These are 1 bit registers.

Let us assume after the application of the `stuck-at-1` fault at $x^{(k-1)}$, x becomes x^*, $z = (x+y) \bmod 2^{(k+1)}$ becomes z^*, and the message bit $\hat{m}$ becomes $\hat{m}^*$. We introduce the following two events:

- **Fault activation**: The injected `stuck-at-1` fault at $x^{(k-1)}$ is active if $x^{(k-1)} = 0$, else it is inactive. If the fault is active, then $x^{*(k-1)} = \overline{x^{(k-1)}}$ (the value of the bit changes from 0 to 1).
- **Fault propagation**: The injected `stuck-at-1` fault at $x^{(k-1)}$ propagates to $z^{(k)}$ if and only if $z^{*(k)} = \overline{z^{(k)}}$. The injected fault propagates to $z^{(k+1)}$ if and only if $z^{*(k+1)} = \overline{z^{(k+1)}}$. A fault propagation happens only if the fault is "active"

The main operation of `A2B` is to compute $z = x+y \bmod 2^{(k+1)}$, and it performs this by utilizing Boolean masking. However, for our purpose, we just need to focus on the addition functionality and the arithmetic shares (x and y). We can write:

$$z^{(j)} = x^{(j)} \oplus y^{(j)} \oplus c^{(j-1)}, \tag{2}$$

where $1 \leq j \leq (k+1)$, $c^{(j-1)}$ is the carry for $(j-1)$-th bit, and $c^{(0)} = 0$. The carry can be written further as:

$$\begin{aligned} c^{(j-1)} &= (x^{(j-1)} \& y^{(j-1)}) \oplus ((x^{(j-1)} \oplus y^{(j-1)}) \& c^{(j-2)}) \\ &= (x^{(j-1)} \& y^{(j-1)}) \oplus (x^{(j-1)} \& c^{(j-2)}) \oplus (y^{(j-1)} \& c^{(j-2)}) \\ &= ((y^{(j-1)} \oplus c^{(j-2)}) \& x^{(j-1)}) \oplus (y^{(j-1)} \& c^{(j-2)}). \end{aligned} \tag{3}$$

The relation between x, y, z and m_1, m_2, $\hat{m}$ is given as:

$$z^{(k+1)} = x^{(k+1)} \oplus y^{(k+1)} \oplus c^{(k)} = m_1 \oplus m_2 = \hat{m}, \tag{4}$$

$$c^{(k)} = ((y^{(k)} \oplus c^{(k-1)}) \& x^{(k)}) \oplus (y^{(k)} \& c^{(k-1)}). \tag{5}$$

Next, we present the following lemmas:

Lemma 1. *If we introduce a* `stuck-at-1` *fault at* $x^{(k-1)}$*, the fault activates with probability* $\frac{1}{2}$*. The activated fault propagates to* $z^{(k)}$ *(i.e.* $z^{*(k)} = \overline{z^{(k)}}$*) if and only if* $z^{(k-1)} = 1$*.*

Proof. The introduced `stuck-at-1` fault at $x^{(k-1)}$ is *active*, *i.e.* $x^{*(k-1)} = \overline{x^{(k-1)}}$, only if $x^{(k-1)}$ is equal to 0. Since x is a random arithmetic share, $x^{(k-1)} = 0$ happens with probability $\frac{1}{2}$. If the fault is active then $x^{*(k-1)} = 1$. From Equation 2, we get that the introduced fault at $x^{(k-1)}$ can only propagate to $z^{(k)}$ through $c^{(k-1)}$. Let us assume c becomes c^* after the fault injection. Now, from Equation 3, we get $c^{*(k-1)} = ((y^{(k-1)} \oplus c^{(k-2)}) \& x^{*(k-1)}) \oplus (y^{(k-1)} \& c^{(k-2)})$. The fault propagation (i.e. $c^{*(k-1)} = \overline{c^{(k-1)}}$) happens only if $(y^{(k-1)} \oplus c^{(k-2)}) = 1$. This implies that one of $y^{(k-1)}$ and $c^{(k-2)}$ is zero, *i.e.* $(y^{(k-1)} \& c^{(k-2)}) = 0$. So, from Equation 2, we get that $z^{(k-1)} = x^{(k-1)} \oplus (y^{(k-1)} \oplus c^{(k-2)}) = 0 \oplus 1 = 1$. This is the unmasked value of the $(k-1)$th bit of the coefficient that we target. The fault injection undoes the masking for this specific bit, thanks to the fault propagation. Overall, the fault reaches to $z^{(k)}$ with probability $\frac{1}{2}$ when $z^{(k-1)} = 1$. □

Lemma 2. *If we introduce a* `stuck-at-1` *fault at* $x^{(k-1)}$*, the fault activates with probability* $\frac{1}{2}$*. The activated fault propagates to* $z^{(k+1)}$*, i.e.* $(z^{*(k+1)} = \overline{z^{(k+1)}})$ *if and only if* $z^{(k-1)} = 1$ *and* $z^{(k)} = 1$*.*

Proof. According to Lemma 1, the fault at $x^{(k-1)}$ is active with probability $\frac{1}{2}$. Also, the activated fault propagates only if $z^{(k-1)} = 1$. Now, $z^{(k+1)} = x^{(k+1)} \oplus y^{(k+1)} \oplus c^{(k)}$. The fault can only reach $z^{(k+1)}$ through $c^{(k)}$. Next we observe that $c^{(k)} = ((y^{(k)} \oplus c^{(k-1)}) \& x^{(k)}) \oplus (y^{(k)} \& c^{(k-1)})$. Therefore, the only path for the fault to reach c^k from $x^{(k-1)}$ is via $c^{(k-1)}$. Now as shown in Lemma 1, $c^{(k-1)}$ gets corrupted (i.e. $c^{(k-1)} \neq c^{*(k-1)}$) only if $x^{(k-1)} = 0$. So we only need to determine when a fault in $c^{(k-1)}$ propagates to $c^{(k)}$. According to the equation of $c^{(k)}$ (Equation 5), this is only possible if $x^{(k)} \oplus y^{(k)} = 1$. Finally, we observe that $c^{(k-1)} = ((y^{(k-1)} \oplus c^{(k-2)}) \& x^{(k-1)}) \oplus (y^{(k-1)} \& c^{(k-2)}) = (1 \& 0) \oplus 0 = 0$ (from the proof of Lemma 1). Therefore, it is evident that $z^{(k)} = x^{(k)} \oplus y^{(k)} \oplus c^{(k-1)} = 1$. This proves the lemma. □

Example: Now, let us provide a very simple example to visualize Lemma 2. Let us consider $k = 2$ (therefore, we have a 3-bit register). In this case, we introduced a `stuck-at-1` fault at $x^{(1)}$. The fault is active only if $x^{(1)}$ is equal to 0. If the fault is active then $z^* = (z+1) \bmod 2^3$. Table 3 presents the values of z and $z^{(3)}$ before the `stuck-at-1` fault at $x^{(1)}$ together with the values of z^* and $z^{*(3)}$ after the fault, when the `stuck-at-1` fault at $x^{(1)}$ is active. From this example, we can observe that the fault at $x^{(1)}$ only propagates to $z^{(3)}$, if $z^{(1)} = 1$ and $z^{(2)} = 1$.

Table 3: The value of z and $z^{(3)}$ before and after the `stuck-at-1` fault at $x^{(1)}$

Before the fault injection				After the fault injection	
$z^{(3)}$	$z^{(2)}$	$z^{(1)}$	z	z^*	$z^{*(3)}$
0	0	0	0	1	0
		1	1	2	0
	1	0	2	3	0
		1	3	4	1
1	0	0	4	5	1
		1	5	6	1
	1	0	6	7	1
		1	7	0	0

Lemma 3. *A `stuck-at-1` fault at $x^{(k-1)}$ activates with probability $\frac{1}{2}$ and propagates to $z^{(k+1)}$ and causes decryption failure if and only if $z^{(k-1)}=1$ and corresponding message bit $\hat{m}=1$.*

Proof. From Equation 1 and Equation 4, we have $z^{(k)}=z^{(k+1)}=\hat{m}$. So, the `stuck-at-1` fault at $x^{(k-1)}$ propagates to $z^{(k+1)}=\hat{m}$, only if $z^{(k-1)}=1$ and the corresponding message bit is 1 (according to the previous lemmas). As the fault propagates to $\hat{m}$ then $\hat{m^*}=\overline{\hat{m}}$. This event will cause decryption failure as the decoded message bit is different before and after the fault injection. Once again we remind that $\hat{m}$ will remain as two shares. However, by the correctness of the Boolean masking, the properties described in these lemmas will follow. □

In a nutshell, fault propagation provides information about $z^{(k-1)}$, and propagates till $z^{(k+1)}$ conditioned on $z^{(k)}$. Further, since $z^{(k)}$ and $z^{(k+1)}$ are the message bits (and assumed known in our attacks), if corrupted, they lead to decryption failure. Overall, we obtain a (unmasked) bit ($z^{(k-1)}$) which exposes partial information about the secret-dependent noise. Notably, we receive similar results if we use `stuck-at-0` fault. However, the fault propagation would happen when $z^{(k-1)}=0$, in that case. However, bit-flip faults do not provide information in this case. For a `bit-flip` fault at $x^{(k-1)}$, it is active for both possible cases, when $x^{(k-1)}=0$ or when $x^{(k-1)}=1$. The injected fault will propagate to $z^{(k)}$ and cause successful decryption failure for 2^{k-1} instances out of 2^k instances when $z^{(k-1)}=1$ and also when $z^{(k-1)}=0$. Hence, successful decryption failure does not provide knowledge regarding $z^{(k-1)}$.

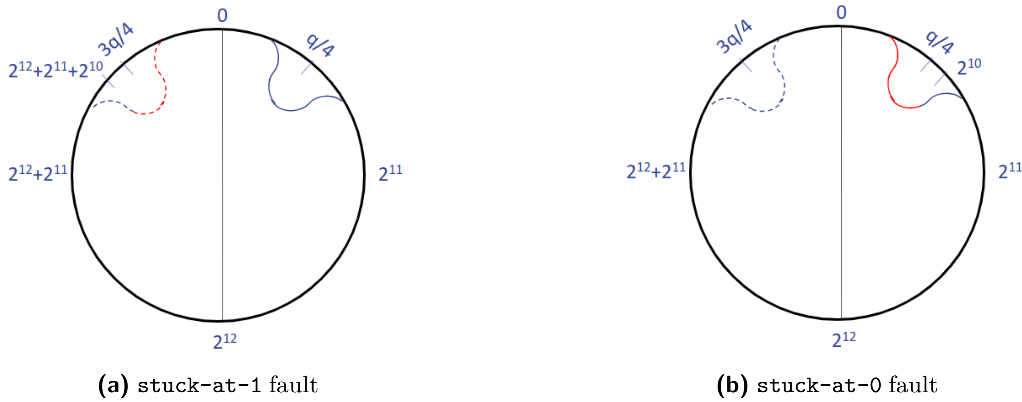
3.2 Fault Propagation for the Higher-Order Masking

We now discuss the effect of our `stuck-at` fault on the higher-order masked decapsulation algorithms. In t-th order masking, the input of `A2B` conversion $g[i]$ has $t+1$ arithmetic shares. For the sake of simplicity, let us assume $z=g[i]$ and its $(t+1)$ arithmetic shares are $(x_1, x_2, ..., x_{t+1})$. So, $(\sum_{j=1}^{(t+1)} x_j) \bmod 2^{(k+1)}=z$. Now, this multi-share sum can be written as a two-share case $z=(x_1+y) \bmod 2^{(k+1)}$, where $y:=(\sum_{j=2}^{(t+1)} x_j) \bmod 2^{(k+1)}$. This brings back the two-share scenario explained in the previous subsection. More precisely, we introduce a `stuck-at-1` fault at x_1^{k-1}. The fault is active only if $x_1^{k-1}=0$. The injected fault at x_1^{k-1} will not affect y, and can propagate to $z^{(k+1)}$ only through x_1^{k-1}. Therefore, by applying Lemma 3, the `stuck-at-1` fault at x_1^{k-1} will propagate to z^{k+1} only if $z^{(k-1)}=1$ and $z^{(k)}=1$. The conclusion is that the fault propagation scenario we describe here does not get affected by the order of masking.

3.3 Application of Our Fault Attack on Kyber

In Kyber, $q=3329$ and hence $k=12$. So, each coefficient of $m' \in \mathbb{R}_q$ are of 12 bits. However, in order to enable message extraction in a masked setting, the shares are transferred from mod q to mod $2^{(k+1)}$, making each share a 13-bit value. *We inject `stuck-at-1` bit-fault at the 11th bit of the i-th coefficient of an input share of the `A2B` of the masked `Decode` algorithm.* Whether or not the fault propagates to the 13-th bit (which remains in a Boolean shared form) exposes the 11-th bit.

According to our nomenclature defined in the previous section, we denote the target 11-th bit (to be faulted) as $g_{a_1}[i]^{(11)}$, and the 11-th-bit value to be extracted as $g[i]^{(11)}$, where $g=(g_{a_1}+g_{a_2}) \bmod 2^{13}$. From Lemma 3, we obtain that the `stuck-at-1` fault at $g_{a_1}[i]^{(11)}$ propagates to the corresponding i-th message bit $m[i]$ and causes decryption failure (with probability 1/2) when $g[i]^{(11)}=1$ and $m[i]=1$. Figure 7a presents the value ranges for $g[i]$, where the `stuck-at-1` fault will propagate and cause decryption failure, and in which cases the fault will not cause decryption failure for Kyber. The value range of $g[i]$, which will cause decryption failure are $\{2^{12}+2^{11}+2^{10}, 2^{12}+2^{11}+2^{10}+1, ..., 2^{13}-1\}$ (coloured in red). The value range of $g[i]$, which will not cause decryption failure are $\{2^{13}-\lceil q/2 \rceil, 2^{13}-\lceil q/2 \rceil+$



(a) `stuck-at-1` fault **(b)** `stuck-at-0` fault

Figure 7: Results of `stuck-at` fault attacks on Kyber. The solid curve represents the probability distribution of a coefficient that decoded to the message value $m[i]=0$ and the dashed curve represents the probability distribution of a coefficient that decoded to the message value $m[i]=1$. This picture also presents that the injected fault will corrupt the message value whenever it lies on the red part.

1, ..., $2^{12}+2^{11}+2^{10}-1\}$(coloured in blue). One important point is that the distributions are bell-shaped, i.e. the mean values occur more often than any other value, and the values at the tails are extremely rare. This fact will be used later for relaxing the fault model to tolerate multi-bit faults. Similarly, the `stuck-at-0` fault at $g_{a_1}[i]^{(11)}$ propagates to $m[i]$ and causes decryption failure when $g[i]^{(11)}=0$ and $m[i]=0$ with probability 1/2. Figure 7b shows the value of $g[i]$ where the `stuck-at-0` fault will propagate and cause decryption failure and where the fault will not cause decryption failure. The values of $g[i]$, which will cause decryption failure are $\{0, 1, ..., 2^{10}-1\}$ (coloured in red) and only when $m[i]=0$ and the fault is active. There are some implementations of masked Kyber where 16 bit register is used for $g[i]$ instead of 13 bit. One such implementation is [HKL+22] (We used this implementation in our practical setup). We note that if the `stuck-at-1` fault at $g[i]^{(11)}$ propagates to $g[i]^{(13)}$ then it will propagate to $g[i]^{(16)}$. This can be proven by repeated application of Lemma 2. Therefore, working with the 13th or the 16th bit is the same for us, and we stick with the 13th bit case.

Now, we recall that the 11th bit corresponds to the (secret dependent) noise. *After the injection of the `stuck-at-1` fault at $g[i]^{(11)}$, it will cause a decryption failure if $g[i]^{(11)}=1$ and the corresponding message bit $m[i]=1$. If $m[i]=1$ and $g[i]^{(11)}=1$, then the associated decryption noise is* $\geq \lfloor q/4 \rfloor - (2^{10} - \lfloor q/4 \rfloor)$. So, it can create a partition in the distribution of the decryption noise depending on the decryption failure. This partition leaks some information regarding the secret key, and we can eventually form inequalities involving the secret key (described in the next subsection). Note that the partition we received from our fault attack is different from the attacks available in the literature [PP21, HPP21, Del22]. In the previous works, if the message bit $m[i]=1$, then after the injection of the fault for almost half of the possible values of $g[i]$ (around $q/4$ values out of $q/2$ values) the fault will cause decryption failure, and for the other half of the possible values of $g[i]$ will not cause decryption failure. The same incident happens when the message bit $m[i]=0$. However, in our `stuck-at-1` attack, we observed decryption failures only when the corresponding message bit $m[i]=1$. If $m[i]=1$, then in our experiments, we have noticed 99.3% decryption failure and only 0.7% decryption success by applying `stuck-at-1` fault at 11th bit of a share of $g[i]$ for Kyber512. We provide more details in Section 5.

3.4 Recovery of the Secret Key for Kyber

This subsection will describe the final stage of our attack *i.e.* recovering the secret key for Kyber. As mentioned earlier, decryption noise forms a linear equation with the secret key $\boldsymbol{s}$ and error $\boldsymbol{e}$. The decryption noise for LWE-based KEM is equal to $\boldsymbol{e}^T\boldsymbol{s'}[i]-\boldsymbol{s}^T(\boldsymbol{e_1}[i]+\Delta\boldsymbol{u}[i])+e_2[i]+\Delta v[i]$, where $\Delta\boldsymbol{u}$ and Δv are the noise introduced because of the compression operation over $\boldsymbol{u}$ and v, respectively. In our case, a decryption failure is observed only when the message bit $m[i]=1$, and the corresponding 11-th bit is also 1. If the fault is active and decryption failure happens, then the decryption noise satisfies the following inequality.

$$(\boldsymbol{e},\boldsymbol{s})^T(\boldsymbol{s'}[i],-(\boldsymbol{e_1}[i]+\Delta\boldsymbol{u}[i]))+e_2[i]+\Delta v[i]\geq\lfloor q/4\rceil-(2^{10}-\lfloor q/4\rceil)\rfloor. \quad (6)$$

If the fault is active and decryption success happens, then the decryption noise satisfies the following inequality.

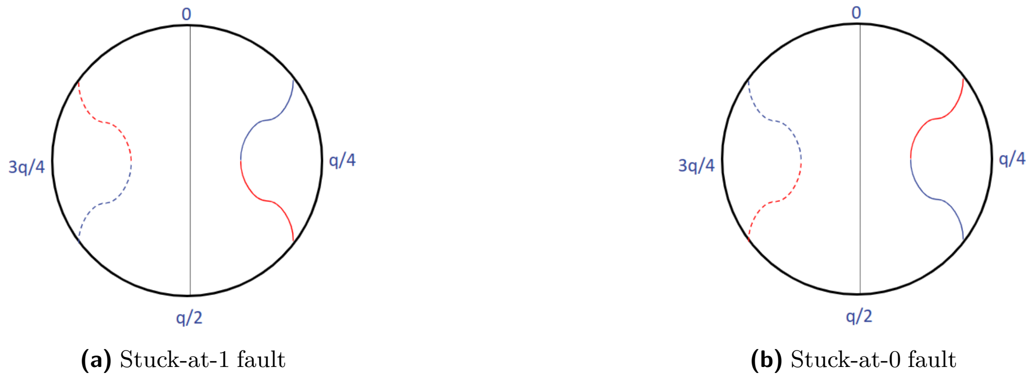
$$(\boldsymbol{e},\boldsymbol{s})^T(\boldsymbol{s'}[i],-(\boldsymbol{e_1}[i]+\Delta\boldsymbol{u}[i]))+e_2[i]+\Delta v[i]<\lfloor q/4\rceil-(2^{10}-\lfloor q/4\rceil). \quad (7)$$

Note that the fault is active with probability $\frac{1}{2}$. As mentioned in Section 1, the attacker has access to the encapsulation oracle and works with self-generated ciphertexts. So, the attacker knows all the values in Equation 6 & 7, except $(\boldsymbol{e},\boldsymbol{s})$. The attacker can find the polynomials $(\boldsymbol{e},\boldsymbol{s})$ by solving these linear inequalities.

Recently, a few inequality solvers have been proposed [PP21, HPP21, Del22, HMS+23]. All these solvers are based on BP in order to tolerate errors and solve the system with a large number of inequalities in a reasonable amount of time. In this work, we have used the inequality solver proposed by Delvaux et al. in [Del22] to solve the inequalities obtained from the `stuck-at-1` fault on Kyber. We always inject the fault at some fixed coefficient. If we consider ω numbers of inequalities, then the system of inequality contains ω inequalities with ψ unknowns. Here the value of ψ is the total number of coefficients of $\boldsymbol{s}$ and $\boldsymbol{e}$, which is $2*l*256$, $l=2$ for Kyber-512. This inequality solver uses an iterative method to solve the system of ω inequalities by maintaining a probability mass function (PMF) for each of ψ unknowns. The PMF for each unknown initiated with the CBD on $[-\eta,\eta]$ and updated in each iteration. The updation of PMF for each unknown is performed by calculating the probability to satisfy each of the ω inequalities for each possible value $[-\eta,\eta]$ of the unknown, and then all the $2*\eta*\omega$ probabilities are combined. [Del22] accelerates the computation time of their solver by applying the central limit theorem. The iteration of the updation of PMF for each unknown continues until a sufficiently approximate one-point distribution on $[-\eta,\eta]$ is found.

3.5 Application of Our `stuck-at` Fault Attack on Saber

In Saber, $q=p=2^{\epsilon_p}$ is a power-of-two moduli and $k=\epsilon_p$ ($\epsilon_p=10$ for the medium security version of Saber). The decoding algorithm in Saber has only two steps. The input of the decoding algorithm of Saber is arithmetic shares of $m'\in\mathbb{R}_p$. First, the algorithm performs the `A2B` conversion on the arithmetic shares of m' to convert them into Boolean shares. Then the most significant bit (ϵ_pth bit) is extracted from the Boolean shares of m' by using (ϵ_p-1) right shift operation. This final output is the message polynomial m, which is the output of the decoding in Saber. $m'[i]$ follows a bell-shaped distribution with a peak at $p/4$ when $m'[i]^{(\epsilon_p)}$ is 0, *i.e.* for the corresponding message bit 0. The values of $m'[i]$ follow a bell-shaped distribution with a peak at $3p/4$ when $m'[i]^{(\epsilon_p)}$ is 1, *i.e.* for the corresponding message bit 1. From Lemma 1, we get that the injected `stuck-at-1` fault at the (ϵ_p-1)-th bit (9th bit for the medium security version of Saber) of a share of $m'[i]$ propagates and causes decryption failure if it is active and $m'[i]\in\{(p/4),(p/4)+1,...,(p/2)-1\}\cup\{3(p/4),3(p/4)+1,...,p-1\}$ (coloured in red). The injected fault is active with 1/2 probability. Also, if $m'[i]\in\{0,1,...,(p/4)-1\}\cup\{(p/2),(p/2)+1,...,3(p/4)-1\}$, then the fault does not propagate and hence does not cause decryption failure (coloured in blue). We use Figure 8a to show the values of $m'[i]$ in which the `stuck-at-1` fault



(a) Stuck-at-1 fault **(b)** Stuck-at-0 fault

Figure 8: Results of `stuck-at` fault attacks on Saber. The probability distribution of $m'[i]$ is shown in these figures. The solid curve represents all the values of $m'[i]$ that decoded to the message value 0, and the dashed curve represents all the values of $m'[i]$ that decoded to the message value 1. The inserted fault will corrupt the message value whenever the value of $m'[i]$ lies on the red part.

propagation is successful and the values of $m'[i]$ in which this fault will not cause decryption failure. In this figure, we use a solid bell curve to denote the distribution for the message bit 0 and a dashed bell curve to indicate the distribution for the message bit 1. Similarly, we get that the injected `stuck-at-0` fault at the $(\epsilon_p - 1)$-th bit of a share of $m'[i]$ propagates and causes decryption failure if it is active and $m'[i] \in \{0, 1, ..., (p/4)-1\} \cup \{(p/2), (p/2)+1, ..., 3(p/4)-1\}$ (coloured in red). Also, if $m'[i] \in \{(p/4), (p/4)+1, ..., (p/2)-1\} \cup \{3(p/4), 3(p/4)+1, ..., p-1\}$ then decryption failure does not arise(coloured in blue). Figure 8b represents the values of $m'[i]$ for both cases when the `stuck-at-0` fault propagation is successful and when fault propagation is not. Overall, we see that the fault propagation and the nature of the information leakage are quite similar to that of Kyber's.This gives enough evidence that our fault attack can be applied to any LWE-based KEM with some minor changes, as the leakage pattern from the masked decode algorithm is similar for all these schemes, such as we can use Kyber's analysis to NewHope [ADPS16] just by changing the parameters. However, we left this for the extended version of this work. Hereon, we shall mainly focus on the Kyber scenario in this paper and develop the BP-based key recovery algorithm for it.

4 Relaxation of the Fault Model

The attack described so far in this paper requires faults to be injected at a specific bit (the 11th bit) of a register. While such a precise injection has been demonstrated previously in literature [DBC+18, BH22, SBBR+20], injection at a specific bit is always probabilistic and depends on the precision of the injection mechanism. Therefore, we can only expect the fault to happen at the 11th bit with some probability p. On the other hand, research has shown that limiting the width of faults to one bit is feasible [GYTS14] even with low-cost injection mechanisms such as clock-glitch. However, the location of the faulted bits is not well-controllable in many situations. Furthermore, with low-cost setups, multi-bit faults also occur quite frequently. Therefore, it is practical to assume a fault model that injects single-bit faults at random locations, or even multi-bit in a register. Depending on the injection mechanism, the fault model may vary. However, `stuck-at` faults can be observed in several practical scenarios, such as clock-glitch or EM. Even in our practical experiments with EM injection described in Section 5, we observed single/multi-bit `stuck-at-1` faults.

We now analyze the effect of such a relaxed fault model (single/multi-bit faults at different locations) assumption on the proposed attack. One should note that the proposed fault attack utilizes the inequalities constructed from the decryption failures/successes. There

is no direct way to distinguish whether or not the fault has been injected at the desired location (11th bit for Kyber) in the case of a decryption failure. Similarly, in the case of a decryption success, it is hard to determine if it is due to the fault at 11-bit or some other fault. The injections at the undesired locations may generate *noisy* equations, hindering the convergence of the BP algorithm to the correct key value. Generally, BP algorithms can tolerate noise only up to a certain extent [HPP21, Del22].

Interestingly, we have observed that our attack is not significantly affected by such injection noise. The *signal*, in our case, is a single-bit injection (`stuck-at-1`) at the 11th bit. The main reason behind such observation is that due to the carry chain structure, many faults do not propagate till the most significant bit of the addition operation. More precisely, the only way to observe a fault is to observe it at the MSB bit, and a fault in some least significant bits can propagate to the MSB only through the carry chain. The carry chain contains several AND operations, which makes the fault propagation conditional on many bits, making it rare. It is also due to the normally distributed data that we are dealing with. The target variable assumes only a specific set of values with high probability among the entire range, which further hinders the fault propagation from most of the least significant bits. The probability of fault propagation is the highest for the two most significant bits, 12th and 13th bit for Kyber, and the second highest is for the next significant bit, 11th bit for Kyber. However, for the lower-order bits, the probability of fault propagation is negligible. In the next few paragraphs, we further elaborate on why the fault propagation probability is low for lower-order bits.

For the time being, let us further narrow down our discussion to Kyber-512, although the conclusions made would be similar for other variants. As already mentioned in Section 3.4, we extract a single-bit of the decryption noise via faults, which provides us with some information on the secret key. For Kyber512, the mean of the decryption noise is 0 associated with any of the message bits. Furthermore, decryption noise remains between $(-\lfloor \frac{q}{4} \rfloor, \lfloor \frac{q}{4} \rfloor)$ with probability $(1 - \frac{1}{2^{139}})$ [BDK+17] (as the failure probability for Kyber512 is 2^{-139}). The decryption noise associated stays in between $(-\lfloor \frac{q}{8} \rfloor, \lfloor \frac{q}{8} \rfloor)$ with probability $(1 - \frac{1}{2^{26}})$. Concretely it means that obtaining any noise value out of this range would happen only once in 67 million executions. From a physical attack perspective, it is a quite low probability. These observations indicate that (due to the bell curve of noise distribution) the noise values are highly likely to belong within a small range[3].

Let us now analyze the distribution of the message coefficients ($g[i]$) just before the extraction of the message using the `A2B` algorithm (i.e. before line 4 in Algorithm 3). These coefficients contain the message added with the decryption noise. At the input of `A2B`, $g[i] \in (0, \lfloor \frac{q}{2} \rfloor) = (0, 1664)$, if the corresponding message bit $m[i] = 0$, and $g[i] \in (2^{13} - \lfloor \frac{q}{2} \rfloor, 2^{13} - 1) = (6528, 8191)$, when the corresponding message bit $m[i] = 1$. These two ranges are clearly disjoint between 1664 and 6528. For the other end (i.e. 0 and 8191), they are quite close as $0 = 8192$ in the ring. The main observation at this point is that *if a fault corrupts the message bit, then it must be strong enough to take the value of $g[i]$ from one range to the other range (e.g. from the value range of message value 0 to the value range of the message value 1).* As we can see, it is indeed possible if the value of $g[i]$ is close to 0 or 8191. However, such values occur with extremely low probability.

Now we consider the following scenarios, which model various single/multi-bit fault models on this variable $g[i]$. For the sake of explanation, we consider the fault width to be 8 bit, in this case, limited to the least significant bits (in fact, we mostly observe such faults in our practical setup). However, the argument is similar for any bit width. We note that the faults are actually injected on one of the shares of $g[i]$. However, the impact is realized on the unmasked $g[i]$ as the shares are added with `A2B`.

- **`stuck-at-1` fault at least significant 8 bits in a share of $g[i]$**: In this case, random

[3] Note that all these analyses are for Kyber512 and the probability and ranges will slightly vary for other security versions of Kyber. However, the overall pattern remains the same thanks to the bell pattern of the distribution.

bits in between 1 to 8 gets faulted to 1 if its value is 0. If the j-th bit is flipped from 0 to 1, then 2^{j-1} gets added with $g[i]$ after the fault. As we are introducing fault at a random share, the maximum added term to $g[i]$, that can occur from `stuck-at-1` fault at least significant 8 bits in a share of $g[i]$, is $(1+2+4+\cdots+128)=255$. This happens when all of the 8 least significant bits in a share are 0, and all of them flip to 1. So the resulting value addition with $g[i]$ is 255.

- **`stuck-at-0` fault at least significant 8 bits in a share of $g[i]$**: Here, random bits in between 1 to 8 get changed from 1 to 0. If the j-th bit is faulted from 1 to 0, then 2^{j-1} gets subtracted from $g[i]$ after the flip (as we are introducing fault at a masking share). So, the maximum subtracted term, that can occur from `stuck-at-0` fault at least significant 8 bits in a share of $g[i]$, is $(1+2+4+\cdots+128)=255$.
- **`bit-flip` fault at least significant 8 bits in a share of $g[i]$**: In this case, random bits in between 1 to 8 get flipped from either 0 to 1 or 1 to 0. If the j-th bit is flipped from 0 to 1, then 2^{j-1} gets added with $g[i]$ after the flip, and if the j-th bit is flipped from 1 to 0, then 2^{j-1} gets subtracted from $g[i]$ after the flip. So, the maximum added term that can occur in this case is 255, as well as it is the maximum subtracted term that can occur.

Let us first consider the scenario when the message bit $m[i]$ associated with $g[i]$ is 1. The decryption failure happens when we can change $g[i]$ to some values that decode to 1, and we want to do it with 8-bit fault models described before. This can indeed happen for $g[i]=8191$ or values closer to it if we add 255 with it. However, the least possible value for which this would be feasible is 7937, which occurs with a very low probability. If we consider values from the higher probability range, such as (6944, 7776) (happens with probability $1-\frac{1}{2^{26}}$), the chances that it will go to the ranges where the message value is 0 by adding (resp, subtracting) 255 is zero. In other words, the probability of changing a message value from 1 to 0 with an 8-bit fault is definitely $< \frac{1}{2^{26}}$, which we consider as quite low in terms of physical attacks. This is illustrated in Table. 4. In conclusion, we point out that with *multi-bit faults at LSB positions up to 8 bits, it is highly improbable to change the value of the message bit. In other words, the faults at lower order bits never reach the 12/13th bit and, therefore, do not cause noise in our observations.*

It is worth mentioning that the probability of fault propagation to the message bit indeed increases if we consider multi-bit faults up to 9th bits. However, in simulation, we observed that the probability of such propagation is still low (e.g. no decryption failure for up to 9th-bit injection even if we simulate for 1 million cases). For faults up to 10/11 bits, the probability is significantly high for random bit-flip faults (nearly 50%), but lower for stuck-at faults (nearly 30%). The 11th bit is our signal and therefore, we always want a single bit fault individually occurring there. This also happens with some probability and in the experimental section, we show that it is indeed possible to increase this probability by careful adjustment of fault parameters. Finally, for 12th and 13th bits, the fault happens with 50% probability and carries no information. The takeaway of this section, therefore, is we have to carefully avoid injecting faults in these MSB bits and keep the width of multi-bit faults low. Interestingly, most of the single-bit faults, even while occurring at random locations other than the 11/12/13th bit, never propagate. Overall, we see that the fault models can be significantly relaxed for our case, and even with a few single bit stuck-at faults hitting the 11th bit, we can perform the desired fault propagation. There are many situations where the probability of noise is low (or, in fact, zero), and if the fault model can be tuned for any one of those situations, the attack would work seamlessly.

5 Experiments

In this section, we validate the attack on Kyber in simulation and also through practical fault injections. We begin with simulating the ideal scenario, i.e. injecting `stuck-at-1` at the 11th bit of an arithmetic share of $g[i]$ for every execution of the decapsulation algorithm. The outcomes are then fed to the BP algorithm to recover the secret key. The simulated

Table 4: The effect of `stuck-at-1`, `stuck-at-0`, and `bit-flip` fault at the least significant 8 bits of $g[i]$

Fault model	Before the fault injection			After the fault injection			Approximate probability of the fault-induced decryption failure
	Message $m[i]$	$g[i]$		$g^*[i]$		Message $m^*[i]$	
		Min	Max	Min	Max		
`stuck-at-1`	1	6944	7776	6944	8031	1	$\frac{1}{2^{26}}$
`stuck-at-0`				6689	7776	1	
`bit-flip`				6689	8031	1	

experiment helps us to find the right parameter settings for the BP algorithm. Next, we move to our practical setup with EM-based fault injections on a software implementation. We further tune the BP algorithm to perform key recovery from these practical fault cases.

5.1 Noise Removal from Inequalities

Given a fault injected in the masked decapsulation algorithm with a ciphertext C, the decryption failure provides the information that $g[i]^{(11)}=1$. The decryption success indicates that $g[i]^{(11)}=0$. *The fault must be active in both cases.* However, in a single execution with C, it might happen that we observe successful decryption even though $g[i]^{(11)}=1$ due to the fact that the fault is not activated. Note that fault activation happens with probability $\frac{1}{2}$ since we fault the masked values. In order to remove the uncertainty (which results in wrong information and is interpreted as noise in the BP algorithm) due to probabilistic fault activation, our strategy is to execute the decapsulation of C multiple times (of course, with fault injection). In our injection campaign, we repeat C, β times until we see a decryption failure for it. In case no decryption failure is observed after β executions, we decide $g[i]^{(11)}=0$. Once we see a decryption failure, we stop the execution with C, conclude $g[i]^{(11)}=1$, and move to the next ciphertext.

In our simulation, we begin with $\beta=20$. This is an empirically chosen value. However, we checked (through simulation) that the probability of getting a wrong decision about the value of $g[i]^{(11)}$ is negligible for this choice of β. Moreover, as mentioned before, the probability of $g[i]^{(11)}=1$ is significantly higher (99.3%) than $g[i]^{(11)}=0$, by the nature of the decryption noise distribution. A result of such bias in the distribution is that we get a very low number of decryption "success" cases compared to the decryption "failure" cases. Since we stop execution for a given C upon seeing a decryption failure, the average number of repetitions we need is quite low compared to β. Empirically, we observed that, on average, we need to repeat our fault injection 2.67 times for each ciphertext (We tried with 100K ciphertext, and the total fault required there was 2.67K). The parameter setting for the ideal simulation is presented in Table 5.

Table 5: The parameter settings of the BP algorithm for recovering the whole secret key

Fault model	Required inequalities	Different ciphertexts	Repetition (β)	Required Faults
Simulated ideal fault model	30,000	60,000	20	160,719
Practical fault model	35,000	70,000	180	1,857,294

5.2 Configuring and Running the BP Algorithm

As mentioned in Section 3.4 we have utilized the inequality solver proposed by [Del22]. In our `stuck-at-1` fault injection attack at $g[i]^{(11)}$, we are only dealing with the inequalities that are generated from the message bit $m[i]=1$. This follows from the Lemmas in Section 3.1, which say that the fault propagates to the 13th bit only when the 12th bit is 1. Since for Kyber,

both the 12th and 13th bit are the same message bit, this means that the message bit must be 1. The corresponding inequalities for decryption success and failures are given in Equation 7 and Equation 6, respectively, in Section 3.4. However, due to such a highly unbalanced value distribution for $g[i]^{(11)}$, the BP solver might end up having a negligible number of inequalities corresponding to the correct cases compared to the faulty cases. The ratio is originally 0.7 : 99.3, which improves to 3 : 97 by applying a filtering strategy described in the next paragraph. This ratio, however, hinders the BP solving, making it converge to a wrong value quite often. One most probable reason for this would be that the highly-biased ratio makes the solver ignore the constraints due to correct cases, and impedes the solver converge to a single key. To stabilize this, we introduce a strategy called *sample rejection. More precisely, we randomly reject* 50% *of the inequalities among the total inequalities generated, but these rejected inequalities are only chosen from the decryption failure cases.* This improves the ratio between correct and faulty cases, making it 7 : 93. We found that this ratio is sufficient for consistently solving the system for several random secret key choices. With this new balanced system of inequalities, we need roughly 30k inequalities to recover the whole secret key (shown in Figure 11), and to obtain these 30k inequalities (i.e. 60k different ciphertexts, as we reject 50% of the total inequalities and all are generated from decryption failures), we require to continue the fault simulation 160k times (= repetitions required for noise removal × ciphertexts required to enable 50% sample rejection = 2.67 × 60k). We present this result in Table 5.

Another important parameter that controls the BP solving is the *filtering* strategy originally presented in [Del22]. Filtering basically chooses the smallest value among the message coefficients (i.e. $g[i]$) for fault injection in order to assist the BP algorithm. Also, in our case, it has some impact on improving the ratio of correct and faulty executions, as mentioned in the last paragraph. While directly implementing this strategy can be costly from the implementation perspective of the injection setup (as we need to change the fault location for every ciphertext), this is functionally similar to the following strategy: fixed the fault location at some coefficient index i, and then choose a ciphertext having the minimum value for this coefficient (among several randomly generated ciphertexts) for fault injection. This strategy, however, increases our simulation time, as we now need to generate several encapsulations to enable the minimum value selection. Owing to the similarity of the original filtering strategy, we, therefore, used the original filtering already implemented in the BP code. However, we take the minimum value for which the message bit is also 1. Also, it is worth mentioning that during the practical attack, we keep the fault location fixed, mimicking the second strategy.

5.3 Practical Setup

After establishing the simulated evidence for the attack, we now move to the practical fault injection experiments. For our experiments, we target the reference first-order masked implementation of the considered scheme built for the ARM Cortex-M4 family of microcontrollers [HKL+22]. We ported the reference implementation to the STM32F4-DISCOVERY board (DUT) housing the STM32F407 – an ARM Cortex-M4 microcontroller running at a clock frequency of 24 MHz. For the compilation of the reference implementation, we have used arm-none-eabi-gcc version 10.3.1 with compiler flags *-O3 -std=gnu99 -mthumb -mcpu=cortex-m4 -mfloat-abi=hard -mfpu=fpv4-sp-d16*. We have used a generic USB 2.0 to TTL converter for USART communication with the DUT. EM Fault Injection (EMFI) was utilized to inject faults into the target device.

The most formidable challenge in executing the aforementioned attack is inducing the faults accurately at desired locations. We have established an injection setup to address this challenge, illustrated in Figure 9. This setup consists of several components, including an arbitrary waveform generator (Keysight 81160A), a constant-gain power amplifier (Teseq CBA 400M-260), a high-frequency near field H-probe (Rigol Near-field Probe 30MHz-3GHz), and an XYZ table (Thorlabs SMC100). Figure 10 shows the position of the near-field EM probe on the DUT.

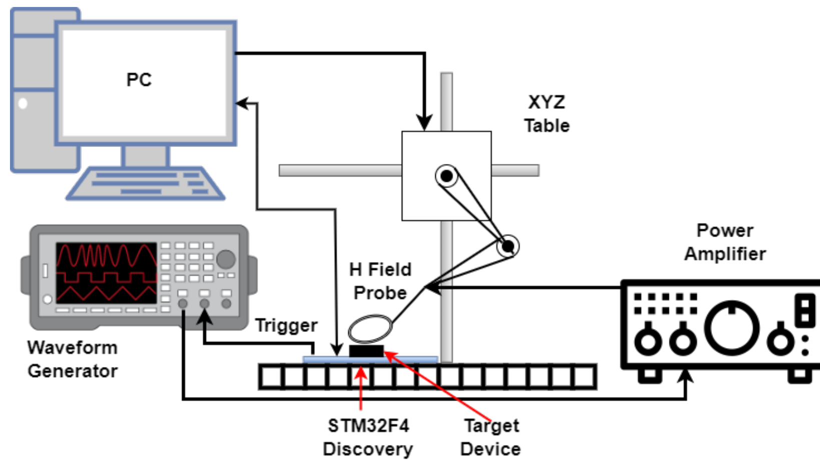


Figure 9: Schematic of the attack platform

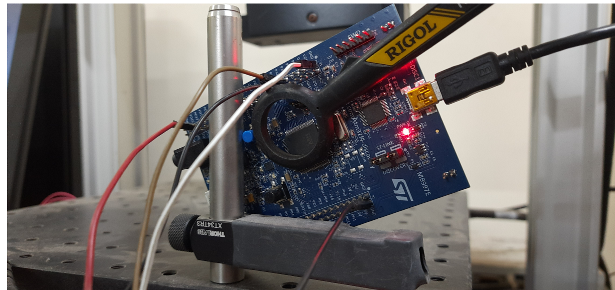


Figure 10: The position of the probe on the STM32F4 Discovery board

After being triggered by a signal from the evaluation board hosting the target implementation, the waveform generator produces a high-frequency pulse train. Following this, the amplifier boosts the intensity of this pulse train, and the H-probe generates a magnetic field above the target. It's important to highlight that the pulse train's amplitude, frequency, and burst count can be customized to meet particular needs. It was observed that by manipulating the position of the probe over the target using the XYZ table, with some necessary adjustments to the pulse parameters, fault can be induced in specific bits of the target register. However, this positioning process must rely on a trial-and-error approach due to the absence of internal register visibility. We have also utilized an oscilloscope to observe the execution timing of the target instruction so that the pulse delays can be adjusted accordingly and the trial-and-error can take the least amount of time. Initially, we analyzed the target object file to get an insight into the occurrence of a particular instruction where we can inject the desired fault. In the *masked_poly_tomsg*() function of the reference implementation, we have targeted a specific arithmetic instruction to induce the fault in the desired register, which corresponds to line 3 of Algorithm 3.

5.4 Attack with Practical Faults

We do not expect the practically injected faults to exactly follow the ideal distribution (i.e. `stuck-at-1` fault at 11th bit with 100% probability). It is more judicious to assume that the desired fault (`stuck-at-1` at $g[i]^{(11)}$) will occur with a certain probability p. If p is significant, then we can expect some *signal* (the desired fault event) to construct useful inequalities. Injections at undesired locations cause noise. However, we can handle such

noise quite well, thanks to the two observations made in this paper: 1) A significant part of the induced faults on a share of $g[i]$ does not propagate to the 12th or the 13th bit (ref. Section 4), and 2) The uncertainty (noise) due to undesired injections can be largely removed through repeated execution of the same ciphertext (Section 5.1). In Section 5.1, the second observation was utilized to remove the uncertainty arising due to fault activation. However, the same strategy can be utilized to remove the uncertainties for undesired fault locations.

We, therefore, follow the same strategy as before: Upon repeating injection several (β) times for the same ciphertext C, if we see a decapsulation failure, we decide $g[i]^{11}=1$. If we do not observe any decapsulation failure, $g[i]^{11}=0$. The choice of β is crucial here, as a very low choice might lead to several wrong inequalities being added to the system. A very high choice, on the other hand, may increase the attack time. We note that even after choosing a proper β, we may end up having some wrong equations. However, the BP algorithm inherently tolerates some (low) noise. Therefore, it does not create an issue for us. The most obvious (albeit slow) way or finding β is to do some trial-and-error. We did this for the first set of outcomes from our practical setup with the same $\beta=20$ value chosen for the ideal simulation. This did not work for obvious reasons. In order to speed up our experiments, we, therefore, extract the fault model directly from the setup, assuming an offline profiling stage. Analyzing faults at the target location, we observed that multi-bit faults were also induced, specifically towards the LSB side of the target register. After tweaking the pulse and burst parameters from the waveform generator, we were able to obtain 10% single-bit stuck-at-1 faults at the 11th bit and 90% random single-bit/multi-bit stuck-at-1 faults, mostly concentrated at the bit positions $1-8$ (the LSB is at position 1). The pulse parameters are presented in Table 6. From this, we empirically estimate the value of β as 180. The interpretation is that if we repeat the fault injection β times for the decapsulation of the same ciphertext, then the probability of the fault getting activated at the 11th bit position at least once is significant. Interestingly, if the faults happen at any of the $1-8$ LSB positions, it does not propagate to the message bit (13th bit of $g[i]$)[4]. Therefore, with this observed fault model, we can ensure that if a decryption failure is observed, it is always due to the propagation of our desired fault. The $g[i]^{(11)}=0$ case, though, may have noise. However, $\beta=180$ successfully removes this noise for most of the cases, enabling key recovery. We notice that, on average, for a single ciphertext, we need to repeat the fault injection 26.53 times only. As shown in Figure 11, we could recover the secret key with 35k inequalities even with this fault model. To obtain 35k different balanced inequalities, we are required to continue the fault simulation 1,857k times (= repetitions required for noise removal $\times$ ciphertexts required to enable 50% sample rejection = 26.53×70k). We present this result in Table 5. It is interesting to observe that the number of inequalities is roughly the same for the ideal and the practical case. This is mainly attributed to the fault model we observe and the potentially removable noise by the structure of the attack.

Table 6: Pulse parameters

Pulse Amplitude	Pulse Frequency	Burst Count
-5.6 dBm	200 MHz	2

6 Discussion on Potential Countermeasures

FA countermeasures use some form of redundant computation to detect/correct the fault [PM18]. Considering the previous attacks on the LWE-based KEMs, many of them can be prevented using this strategy. For example, consider the attack due to [PP21], which skips a subtraction operation. Duplication of the subtraction followed by a check can prevent such attacks. Quite similarly, the attack due to [PP21] can be prevented by duplicating different

[4] This is the reason why we mainly describe the 8-bit fault case in Section 4.

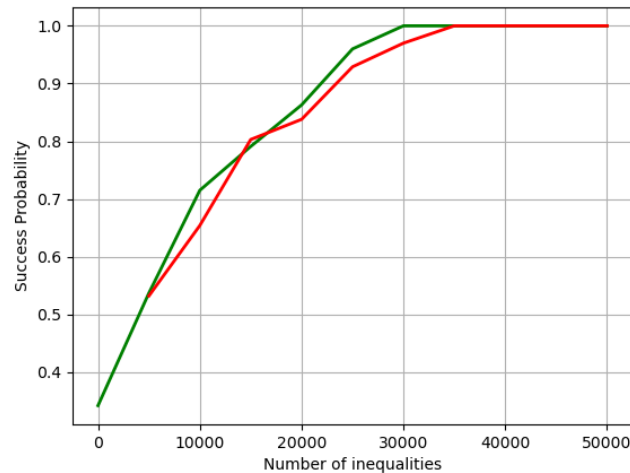


Figure 11: Number of inequalities required to recover partial or entire secret key for Kyber-512 with our simulated and practical fault model. Here, success probability indicates the number of coefficients recovered out of $1024(=512+512)$ coefficients of the secret key. (Success probability equal to 1 means full secret key recovery.) The green coloured curve represents the simulated fault model, and the red coloured line represents the practical fault model.

parts of the re-encryption module as the fault is mainly injected at that part. The reason why duplication works for these cases is that the attacks strongly rely on the `Decode` or the final check operation of the FO transform. Unless the fault reaches these operations, it cannot be exploited. Therefore, preventing the fault from reaching these operations can stop the attack.

In this regard, the proposed attack has some interesting properties. Although we still target the `Decode`, our injection happens much later than the fault location of the attack in [PP21]. In some sense, we target the penultimate step of the decoding operation itself rather than sending some faulty value to decode as in [HPP21, Del22]. The most important fact is that if duplication and detection are used to detect our fault injection, that would result in the same information leakage that we utilize. Error correction might be somewhat useful, however, as shown in recent work [SBJ+21], error correction too leaks if the FA is combined with SCA. We anticipate that recent combined-attack secure gadgets [FGM+23] can be a proper direction to prevent this attack. However, such gadgets are based on Boolean or polynomial masking, while we target arithmetic masking. Shuffling can be useful in this context, but, again, the adversary can perform a combined attack to extract the permutation used for shuffling and thereby remove the noise induced by it. Note that, we already consider masking (even higher-order) for our target implementations. So, extracting only the shuffling permutation would not result in a successful SCA attack. But such extraction would definitely help our FA. With all these observations, we leave a sound countermeasure development as an interesting future work.

7 Conclusions

In this work, we propose a new FA on the SCA-secure masked decapsulation algorithm for generic LWE-based KEMs and elaborate the attack for Kyber. Our attack exploits the `A2B` component used in the masked implementations and appears due to the presence of masking. So, it can be considered as a vulnerability introduced from the algorithmic changes introduced due to masking. The attack results due to fault propagation through the `A2B` component and leaks a secret-dependent noise bit by unmasking it due to faults. Eventually, we show a practical validation of the attack on ARM Cortex-M4 microcontrollers using

EMFI. A direct consequence of the attack is that one must be careful while designing masking algorithms. One also has to be careful while introducing fault countermeasures to prevent this attack, as we anticipate that state-of-the-art duplication-based countermeasures would not be sufficient. Overall, our work encourages more study in this direction, in general, to eventually construct truly secure implementations.

Acknowledgements

This work was partially supported by Horizon 2020 ERC Advanced Grant (101020005 Belfort), CyberSecurity Research Flanders with reference number VR20192203, BE QCI: Belgian-QCI (3E230370) (see beqci.eu), and Intel Corporation. Angshuman Karmakar is funded by FWO (Research Foundation – Flanders) as a junior post-doctoral fellow (contract number 203056 / 1241722N LV). Debdeep would like to thank the ASEM DUO fellowship and the project entitled, "Secure Implementation of Post-Quantum Cryptosystems (SECPQC) TPN NO:- 71447", DST India and BELSPO for partial support.

References

[AAC+22] Gorjan Alagic, Daniel Apon, David Cooper, Quynh Dang, Thinh Dang, John Kelsey, Jacob Lichtinger, Yi-Kai Liu, Carl Miller, Dustin Moody, Rene Peralta, Ray Perlner, Angela Robinson, and Daniel Smith-Tone. Status Report on the Third Round of the NIST Post-Quantum Cryptography Standardization Process. Online. Accessed 26th June, 2023, 2022.

[ADPS16] Erdem Alkim, Léo Ducas, Thomas Pöppelmann, and Peter Schwabe. Post-quantum Key Exchange - A New Hope. In Thorsten Holz and Stefan Savage, editors, *25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016*, pages 327–343. USENIX Association, 2016.

[BDK+17] Joppe Bos, Leo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, John M. Schanck, Peter Schwabe, Gregor Seiler, and Damien Stehlé. CRYSTALS – Kyber: a CCA-secure module-lattice-based KEM. Cryptology ePrint Archive, Report 2017/634, 2017. https://ia.cr/2017/634.

[BDK+20] Michiel Van Beirendonck, Jan-Pieter D'Anvers, Angshuman Karmakar, Josep Balasch, and Ingrid Verbauwhede. A Side-Channel Resistant Implementation of SABER. Cryptology ePrint Archive, Report 2020/733, 2020. https://ia.cr/2020/733.

[BGR+21] Joppe W. Bos, Marc Gourjon, Joost Renes, Tobias Schneider, and Christine van Vredendaal. Masking Kyber: First- and Higher-Order Implementations. *IACR Cryptol. ePrint Arch.*, page 483, 2021.

[BH22] Jakub Breier and Xiaolu Hou. How practical are fault injection attacks, really? *IEEE Access*, 10:113122–113130, 2022.

[BMR21] Luk Bettale, Simon Montoya, and Guénaël Renault. Safe-error analysis of post-quantum cryptography mechanisms - short paper-. In *18th Workshop on Fault Detection and Tolerance in Cryptography, FDTC 2021, Milan, Italy, September 17, 2021*, pages 39–44. IEEE, 2021.

[CGV14] Jean-Sébastien Coron, Johann Großschädl, and Praveen Kumar Vadnala. Secure conversion between boolean and arithmetic masking of any order. In Lejla Batina and Matthew Robshaw, editors, *Cryptographic Hardware and*

Embedded Systems - CHES 2014 - 16th International Workshop, Busan, South Korea, September 23-26, 2014. Proceedings, volume 8731 of *Lecture Notes in Computer Science*, pages 188–205. Springer, 2014.

[CJRR99] Suresh Chari, Charanjit S. Jutla, Josyula R. Rao, and Pankaj Rohatgi. Towards sound approaches to counteract power-analysis attacks. In Michael Wiener, editor, *Advances in Cryptology — CRYPTO' 99*, pages 398–412, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg.

[DBC+18] Jean-Max Dutertre, Vincent Beroulle, Philippe Candelier, Stephan De Castro, Louis-Barthelemy Faber, Marie-Lise Flottes, Philippe Gendrier, David Hély, Regis Leveugle, Paolo Maistri, et al. Laser fault injection at the cmos 28 nm technology node: an analysis of the fault model. In *2018 Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*, pages 1–6. IEEE, 2018.

[Del22] Jeroen Delvaux. Roulette: A diverse family of feasible fault attacks on masked kyber. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2022(4):637–660, 2022.

[DKRV18] Jan-Pieter D'Anvers, Angshuman Karmakar, Sujoy Sinha Roy, and Frederik Vercauteren. Saber: Module-LWR based key exchange, CPA-secure encryption and CCA-secure KEM. Cryptology ePrint Archive, Paper 2018/230, 2018. https://eprint.iacr.org/2018/230.

[FGM+23] Jakob Feldtkeller, Tim Güneysu, Thorben Moos, Jan Richter-Brockmann, Sayandeep Saha, Pascal Sasdrich, and François-Xavier Standaert. Combined private circuits - combined security refurbished. *IACR Cryptol. ePrint Arch.*, page 1341, 2023.

[FO99] Eiichiro Fujisaki and Tatsuaki Okamoto. How to enhance the security of public-key encryption at minimum cost. In *Public Key Cryptography*, pages 53–68, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg.

[FRVD08] Pierre-Alain Fouque, Denis Réal, Frédéric Valette, and M'hamed Drissi. The carry leakage on the randomized exponent countermeasure. In Elisabeth Oswald and Pankaj Rohatgi, editors, *Cryptographic Hardware and Embedded Systems - CHES 2008, 10th International Workshop, Washington, D.C., USA, August 10-13, 2008. Proceedings*, volume 5154 of *Lecture Notes in Computer Science*, pages 198–213. Springer, 2008.

[GYTS14] Nahid Farhady Ghalaty, Bilgiday Yuce, Mostafa Taha, and Patrick Schaumont. Differential fault intensity analysis. In *2014 Workshop on Fault Diagnosis and Tolerance in Cryptography*, pages 49–58. IEEE, 2014.

[HKL+22] Daniel Heinz, Matthias J. Kannwischer, Georg Land, Thomas Pöppelmann, Peter Schwabe, and Daan Sprenkels. First-order masked kyber on ARM cortex-m4. *IACR Cryptol. ePrint Arch.*, page 58, 2022.

[HMS+23] Julius Hermelink, Erik Mårtensson, Simona Samardjiska, Peter Pessl, and Gabi Dreo Rodosek. Belief propagation meets lattice reduction: Security estimates for error-tolerant key recovery from decryption errors. *IACR Cryptol. ePrint Arch.*, page 98, 2023.

[HPP21] Julius Hermelink, Peter Pessl, and Thomas Pöppelmann. Fault-enabled chosen-ciphertext attacks on kyber. In Avishek Adhikari, Ralf Küsters, and Bart Preneel, editors, *Progress in Cryptology - INDOCRYPT 2021 - 22nd International Conference on Cryptology in India, Jaipur, India, December 12-15, 2021, Proceedings*, volume 13143 of *Lecture Notes in Computer Science*, pages 311–334. Springer, 2021.

[JZC+17] Haodong Jiang, Zhenfeng Zhang, Long Chen, Hong Wang, and Zhi Ma. Post-quantum ind-cca-secure kem without additional hash. Cryptology ePrint Archive, Report 2017/1096, 2017. https://eprint.iacr.org/2017/1096.

[KDVB+22] Suparna Kundu, Jan-Pieter D'Anvers, Michiel Van Beirendonck, Angshuman Karmakar, and Ingrid Verbauwhede. Higher-order masked saber. In Clemente Galdi and Stanislaw Jarecki, editors, *Security and Cryptography for Networks*, pages 93–116, Cham, 2022. Springer International Publishing.

[KPP20] Matthias J. Kannwischer, Peter Pessl, and Robert Primas. Single-trace attacks on keccak. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2020(3):243–268, 2020.

[LPR10] Vadim Lyubashevsky, Chris Peikert, and Oded Regev. On ideal lattices and learning with errors over rings. In Henri Gilbert, editor, *Advances in Cryptology - EUROCRYPT 2010, 29th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Monaco / French Riviera, May 30 - June 3, 2010. Proceedings*, volume 6110 of *Lecture Notes in Computer Science*, pages 1–23. Springer, 2010.

[MBB+22] Catinca Mujdei, Arthur Beckers, Jose Bermundo, Angshuman Karmakar, Lennert Wouters, and Ingrid Verbauwhede. Side-channel analysis of lattice-based post-quantum cryptography: Exploiting polynomial multiplication. *IACR Cryptol. ePrint Arch.*, page 474, 2022.

[NDGJ21] Kalle Ngo, Elena Dubrova, Qian Guo, and Thomas Johansson. A side-channel attack on a masked IND-CCA secure saber KEM implementation. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2021(4):676–707, 2021.

[NWDP22] Kalle Ngo, Ruize Wang, Elena Dubrova, and Nils Paulsrud. Side-channel attacks on lattice-based kems are not prevented by higher-order masking. *IACR Cryptol. ePrint Arch.*, page 919, 2022.

[OSPG18] Tobias Oder, Tobias Schneider, Thomas Pöppelmann, and Tim Güneysu. Practical CCA2-Secure and Masked Ring-LWE Implementation. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2018(1):142–174, 2018.

[PM18] Sikhar Patranabis and Debdeep Mukhopadhyay. *Introduction to Fault Attacks*, pages 3–8. Springer Singapore, Singapore, 2018.

[PP19] Peter Pessl and Robert Primas. More practical single-trace attacks on the number theoretic transform. In Peter Schwabe and Nicolas Thériault, editors, *Progress in Cryptology - LATINCRYPT 2019 - 6th International Conference on Cryptology and Information Security in Latin America, Santiago de Chile, Chile, October 2-4, 2019, Proceedings*, volume 11774 of *Lecture Notes in Computer Science*, pages 130–149. Springer, 2019.

[PP21] Peter Pessl and Lukas Prokop. Fault attacks on cca-secure lattice kems. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2021(2):37–60, 2021.

[Reg09] Oded Regev. On lattices, learning with errors, random linear codes, and cryptography. *J. ACM*, 56(6):34:1–34:40, 2009.

[RRB+19] Prasanna Ravi, Debapriya Basu Roy, Shivam Bhasin, Anupam Chattopadhyay, and Debdeep Mukhopadhyay. Number "not used" once - practical fault attack on pqm4 implementations of NIST candidates. In Ilia Polian and Marc Stöttinger, editors, *Constructive Side-Channel Analysis and Secure Design - 10th International Workshop, COSADE 2019, Darmstadt, Germany, April*

3-5, 2019, Proceedings, volume 11421 of *Lecture Notes in Computer Science*, pages 232–250. Springer, 2019.

[RRCB20] Prasanna Ravi, Sujoy Sinha Roy, Anupam Chattopadhyay, and Shivam Bhasin. Generic Side-channel attacks on CCA-secure lattice-based PKE and KEMs. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2020(3):307–335, 2020.

[RSA78] Ronald L. Rivest, Adi Shamir, and Leonard M. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Commun. ACM*, 21(2):120–126, 1978.

[RYB+23] Prasanna Ravi, Bolin Yang, Shivam Bhasin, Fan Zhang, and Anupam Chattopadhyay. Fiddling the twiddle constants - fault injection analysis of the number theoretic transform. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2023(2):447–481, 2023.

[SBBR+20] Sayandeep Saha, Arnab Bag, Debapriya Basu Roy, Sikhar Patranabis, and Debdeep Mukhopadhyay. Fault template attacks on block ciphers exploiting fault propagation. In *Advances in Cryptology–EUROCRYPT 2020: 39th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Zagreb, Croatia, May 10–14, 2020, Proceedings, Part I 39*, pages 612–643. Springer, 2020.

[SBJ+21] Sayandeep Saha, Arnab Bag, Dirmanto Jap, Debdeep Mukhopadhyay, and Shivam Bhasin. Divided we stand, united we fall: Security analysis of some sca+ sifa countermeasures against sca-enhanced fault template attacks. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 62–94. Springer, 2021.

[SJR+19] Sayandeep Saha, Dirmanto Jap, Debapriya Basu Roy, Avik Chakraborty, Shivam Bhasin, and Debdeep Mukhopadhyay. A framework to counter statistical ineffective fault analysis of block ciphers using domain transformation and error correction. *IEEE Transactions on Information Forensics and Security*, 15:1905–1919, 2019.

[SRJB23] Sayandeep Saha, Prasanna Ravi, Dirmanto Jap, and Shivam Bhasin. Non-profiled side-channel assisted fault attack: A case study on domrep. In *2023 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1–6. IEEE, 2023.

[TMA11] Michael Tunstall, Debdeep Mukhopadhyay, and Subidh Ali. Differential fault analysis of the advanced encryption standard using a single fault. In Claudio A. Ardagna and Jianying Zhou, editors, *Information Security Theory and Practice. Security and Privacy of Mobile Devices in Wireless Communication - 5th IFIP WG 11.2 International Workshop, WISTP 2011, Heraklion, Crete, Greece, June 1-3, 2011. Proceedings*, volume 6633 of *Lecture Notes in Computer Science*, pages 224–233. Springer, 2011.

[VOGR18] Felipe Valencia, Tobias Oder, Tim Güneysu, and Francesco Regazzoni. Exploring the vulnerability of R-LWE encryption to fault attacks. In John Goodacre, Mikel Luján, Giovanni Agosta, Alessandro Barenghi, Israel Koren, and Gerardo Pelosi, editors, *Proceedings of the Fifth Workshop on Cryptography and Security in Computing Systems, CS2 2018, Manchester, United Kingdom, January 24, 2018*, pages 7–12. ACM, 2018.