

Roulette: A Diverse Family of Feasible Fault Attacks on Masked Kyber*

Jeroen Delvaux 

Cryptography Research Centre, Technology Innovation Institute, Abu Dhabi, UAE

jeroen.delvaux@tii.ae

Abstract. At Indocrypt 2021, Hermelink, Pessl, and Pöppelmann presented a *fault attack* against KYBER in which a system of linear inequalities over the private key is generated and solved. The attack requires a laser and is, understandably, demonstrated with simulations—not actual equipment. We facilitate and diversify the attack in four ways, thereby admitting cheaper and more forgiving fault-injection setups. Firstly, the *attack surface* is enlarged: originally, the two input operands of the ciphertext comparison are covered, and we additionally cover re-encryption modules such as binomial sampling and butterflies in the last layer of the *inverse number-theoretic transform* (INTT). This extra surface also allows an attacker to bypass the custom countermeasure that was proposed in the Indocrypt paper. Secondly, the *fault model* is relaxed: originally, precise bit flips are required, and we additionally support set-to-0 faults, random faults, arbitrary bit flips, and instruction skips. Thirdly, masking and blinding methods that randomize intermediate variables kindly help our attack, whereas the IndoCrypt attack is like most other fault attacks either hindered or unaltered by countermeasures against passive *side-channel analysis* (SCA). Randomization helps because we randomly fault intermediate prime-field elements until a desired set of values is hit. If these prime-field elements are represented on a circle, which is a common visualization, our attack is analogous to spinning a *roulette* wheel until the ball lands in a desired set of pockets. Hence, the nickname. Fourthly, we accelerate and improve the error tolerance of solving the system of linear inequalities: run times of roughly 100 minutes are reduced to roughly one minute, and inequality error rates of roughly 1% are relaxed to roughly 25%. Benefiting from the four advances above, we use a reasonably priced ChipWhisperer[®] board to break a masked implementation of KYBER running on an ARM Cortex-M4 through *clock glitching*.

Keywords: Fault Attack · Kyber · Key-Encapsulation Mechanism · Lattice-Based Cryptography · Post-Quantum Cryptography

1 Introduction

KYBER [ABD⁺20] is a lattice-based *key-encapsulation mechanism* (KEM) and was selected as a *post-quantum cryptography* (PQC) standard by the United States' *National Institute of Standards and Technology* (NIST) in July 2022. We revisit a fault attack against KYBER proposed by Hermelink, Pessl, and Pöppelmann at Indocrypt 2021 [HPP21a], where a single ciphertext bit in either input operand of the ciphertext comparison must be flipped. Every faulted decapsulation provides one inequality over the private key, and fewer than 10000 inequalities suffice to break all versions of KYBER.

*This paper extends an article accepted to CHES 2022. The only legitimate source for downloading this extended version is the Cryptology ePrint Archive, <https://eprint.iacr.org/2021/1622>.

Hermelink et al. [HPP21a] suggest using a laser to flip a single bit; lasers have a small *spot size* and, therefore, attain a high spatial precision. The time, budget, and expertise needed to (i) decapsulate a target chip [SH07] and (ii) calibrate and use such a specialized fault-injection setup is substantial—but unaccounted for. Especially in scenarios where a single device instead of a batch of devices is targeted, the time spent on building the setup has no advantages of scale and likely surpasses the time needed for the actual key-recovery. Presumably for the above reasons, the authors simulate the use of a laser in software.

1.1 Contributions

We improve the practicality of the above fault attack such that even a low-budget adversary has plenty of options. Four advances are made:

- Before the IndoCrypt paper [HPP21a], the ciphertext comparison was already identified as a prime target for fault attacks [OSPG18, XIU⁺21]. We forewarn secure-system designers that previously untargeted building blocks of the re-encryption should be protected against fault attacks too. This includes binomial sampling, butterflies in the last layer of the *inverse number-theoretic transform* (INTT), ciphertext compression, and its preceding modular reduction. By faulting any of these building blocks, an attacker can obtain inequalities over the private key while bypassing any potential countermeasures that guard the ciphertext comparison. One such countermeasure is proposed in the IndoCrypt paper.
- Whilst the IndoCrypt attack [HPP21a] requires a laser to precisely flip a bit, we support various equipment through various fault models, i.e., set-to-0 faults, set-to-1 faults, random faults, arbitrary bit-flip patterns, instruction skips, and instruction corruptions. The flip side is that more faults are needed for a key recovery: roughly speaking, 1000s become 10000s or 100000s. Even so, for a KYBER implementation that is clocked at MHz rates, and depending on its efficiency and security level, the latter range typically equates to a few hours up to a few days and thus a feasible attack. Furthermore, the additional time needed for a key recovery can partially, if not completely, be recouped by not having to set-up and calibrate a laser. This thought actually pertains to the entire field of study: in many papers that propose fault attacks using pure theory and no equipment, minimizing the number of faults is the exclusive focus [ASMM18], i.e., penalties encountered in practice and caused by strong theoretical assumptions are missing from the optimization model.
- Because most building blocks of KYBER have known weaknesses against *side-channel analysis* (SCA), such as power-consumption analysis, countermeasures should be in place [RCB22]. We lay out a peculiar case where masking and blinding methods that randomize intermediate variables facilitate a fault attack. Under normal circumstances, which includes the IndoCrypt paper [HPP21a], the vulnerability to fault attacks either decreases or remains the same upon introducing these countermeasures. We fault otherwise input-defined prime-field elements such that they cover a wide range of values, ideally but not necessarily uniformly distributed, so data-randomizing countermeasures naturally help achieving a more uniform coverage. To succeed, an attacker needs to keep faulting the element until its value is contained in a specific subset of values. In related work, field elements are often represented on a circle [OSPG18], or in our analogy, a wheel from the casino game roulette. Every fault spins the wheel until, eventually, the ball lands in a winning set of pockets.
- The IndoCrypt paper [HPP21a] presents an algorithm based on *belief propagation* to solve systems of linear inequalities. Solving 7000 inequalities for KYBER768 takes approximately 100 minutes using a single thread. To get around this inconvenience,

the authors parallelize their code: 32 threads on 16 cores result in circa 7 minutes. Instead, we deploy an accurate numerical approximation that reduces the execution time to roughly one minute using a single thread. Upscaling the hardware through threading remains possible but is no longer needed. A second, more acute problem with the solver from the IndoCrypt paper is that all inequalities are assumed to be correct, but fault-injection setups that supposedly provide these inequalities are not perfectly reliable. Based on a previous report by Pessl and Prokop [PP21a], a 1% error rate is yet to be exceeded. We alter the algorithm such that at least 25% of the inequalities can be incorrect. To tie the above two improvements together: higher error rates necessitate more inequalities and thus more computation time, causing our acceleration technique to pay off. Our solver is made open-source.

To demonstrate the above four advances, we break a masked implementation of KYBER running on an ARM Cortex-M4. A ChipWhisperer[®] board, which is affordable for individuals not just organizations, is used to inject faults in the INTT through clock glitching, thereby providing inequalities that are mostly but not always correct.

1.2 Structure

The remainder of this paper is structured as follows. Sections 2 to 4 provide preliminaries on KYBER, SCA, and fault attacks respectively. Section 5 presents our roulette attacks from a theoretical perspective. Section 6 presents our solver. Section 7 presents ChipWhisperer experiments. Section 8 concludes this work.

1.3 Notation

Variables and constants are denoted by characters from the Latin and Greek alphabets respectively. Vectors and matrices are denoted by bold lowercase and bold uppercase characters respectively. Functions are printed in a sans-serif font. Operator $\lceil \cdot \rceil$ denotes rounding to the nearest integer where ties, i.e., fractions of exactly 0.5, are rounded up.

2 Kyber

KYBER [ABD⁺20] starts from a *public-key encryption* (PKE) scheme that is secure against *chosen-plaintext attacks* (CPAs), as recapitulated in Section 2.1, and to which a variation of the *Fujisaki–Okamoto* (FO) transform is applied to additionally resist *chosen-ciphertext attacks* (CCAs), as summarized in Section 2.2. We abstain from comprehensive descriptions and only highlight aspects that are important for this work.

2.1 Public-Key Encryption

The PKE scheme consists of key generation, encryption, and decryption, as specified in Algorithms 1 to 3 respectively. For brevity, the use of binary encodings to efficiently transmit data is omitted. Parameters corresponding to three security levels are given in Table 1. The security of the scheme is based on the *module learning with errors* (MLWE) problem. Errors are drawn from a *centered binomial distribution* (CBD), i.e., $e \triangleq e_1 - e_2$ where $e_1, e_2 \sim B(\epsilon, 1/2)$.

Polynomial arithmetic is performed in the ring $\mathbb{Z}_\rho[x]/(x^\eta + 1)$, where degree $\eta = 256$ of the irreducible polynomial is a power of two and where prime $\rho = 3329 = 256 \cdot 13 + 1$ so that the η -th root of unity exists, i.e., $\zeta^{256} \bmod \rho = 1$ where $\zeta = 17$. These design choices allow polynomial multiplications to be realized with quasilinear time complexity $\mathcal{O}(\eta \cdot \log_2 \eta)$ through the *number-theoretic transform* (NTT) according to Eq. (1), where operator \circ comprises $\eta/2 = 128$ products of linear polynomials.

Algorithm 1 Kyber.PKE.KeyGen

Output: Public key k_{pub}
Output: Private key \hat{s}

- 1: $d \leftarrow \{0, 1\}^{256}$
- 2: $(q, b) \leftarrow H_1(d)$
- 3: **for** $i \in [0, \kappa - 1]$ **do**
- 4: **for** $j \in [0, \kappa - 1]$ **do**
- 5: $\hat{A}[i, j] \leftarrow \text{Parse}(\text{XOF}(q; j, i))$
- 6: $s[i] \leftarrow \text{CBD}(\text{PRF}(b; i); \epsilon_1)$
- 7: $\hat{s}[i] \leftarrow \text{NTT}(s[i])$
- 8: $e[i] \leftarrow \text{CBD}(\text{PRF}(b; \kappa + i); \epsilon_1)$
- 9: $\hat{e}[i] \leftarrow \text{NTT}(e[i])$

- 10: $\hat{t} \leftarrow \hat{A} \circ \hat{s} + \hat{e}$
- 11: $k_{\text{pub}} \leftarrow \hat{t} \| q$

Algorithm 2 Kyber.PKE.Encrypt

Input: Public key $k_{\text{pub}} \triangleq \hat{t} \| q$
Input: Message m
Input: Coins r
Output: Ciphertext $c \triangleq (u, v)$

- 1: **for** $i \in [0, \kappa - 1]$ **do**
- 2: **for** $j \in [0, \kappa - 1]$ **do**
- 3: $\hat{A}^\top[i, j] \leftarrow \text{Parse}(\text{XOF}(q; i, j))$
- 4: $r[i] \leftarrow \text{CBD}(\text{PRF}(r; i); \epsilon_1)$
- 5: $\hat{r}[i] \leftarrow \text{NTT}(r[i])$
- 6: $e_1 \leftarrow \text{CBD}(\text{PRF}(r; \kappa + i); \epsilon_2)$
- 7: $e_2 \leftarrow \text{CBD}(\text{PRF}(r; 2\kappa); \epsilon_2)$
- 8: $\hat{u} \leftarrow \hat{A}^\top \circ \hat{r}$
- 9: **for** $i \in [0, \kappa - 1]$ **do**
- 10: $u'[i] \leftarrow \text{INTT}(\hat{u}[i]) + e_1$
- 11: $u[i] \leftarrow \text{Compress}(u'[i]; \rho, \delta_u)$
- 12: $m' \leftarrow \text{Decompress}(m; \rho, 1)$
- 13: $v' \leftarrow \text{INTT}(\hat{t}^\top \circ \hat{r}) + e_2 + m'$
- 14: $v \leftarrow \text{Compress}(v'; \rho, \delta_v)$

Algorithm 3 Kyber.PKE.Decrypt

Input: Private key \hat{s}
Input: Ciphertext $c \triangleq (u, v)$
Output: Message m

- 1: **for** $i \in [0, \kappa - 1]$ **do**
- 2: $u'[i] \leftarrow \text{Decompress}(u[i]; \rho, \delta_u)$
- 3: $\hat{u}[i] \leftarrow \text{NTT}(u'[i])$
- 4: $v' \leftarrow \text{Decompress}(v; \rho, \delta_v)$
- 5: $m' \leftarrow v' - \text{INTT}(\hat{s}^\top \circ \hat{u})$
- 6: $m \leftarrow \text{Compress}(m'; \rho, 1)$

Algorithm 4 Kyber.KEM.KeyGen

Output: Public key k_{pub}
Output: Private key k_{priv}

- 1: $z \leftarrow \{0, 1\}^{256}$
- 2: $(k_{\text{pub}}, \hat{s}) \leftarrow \text{Kyber.PKE.KeyGen}()$
- 3: $h \leftarrow H_2(k_{\text{pub}})$
- 4: $k_{\text{priv}} \leftarrow \hat{s} \| k_{\text{pub}} \| h \| z$

Algorithm 5 Kyber.KEM.Encapsulate

Input: Public key k_{pub}
Output: Ciphertext c
Output: Symmetric key k

- 1: $m \leftarrow \{0, 1\}^{256}$
- 2: $m \leftarrow H_2(m)$
- 3: $(k', r) \leftarrow H_1(m \| H_2(k_{\text{pub}}))$
- 4: $c \leftarrow \text{Kyber.PKE.Encrypt}(k_{\text{pub}}, m, r)$
- 5: $k \leftarrow \text{KDF}(k' \| H_2(c))$

Algorithm 6 Kyber.KEM.Decapsulate

Input: Ciphertext c
Input: Private key $k_{\text{priv}} \triangleq \hat{s} \| k_{\text{pub}} \| h \| z$
Output: Symmetric key k

- 1: $m \leftarrow \text{Kyber.PKE.Decrypt}(\hat{s}, c)$
- 2: $(k', r) \leftarrow H_1(m \| h)$
- 3: $c' \leftarrow \text{Kyber.PKE.Encrypt}(k_{\text{pub}}, m, r)$
- 4: **if** $c = c'$ **then**
- 5: $k \leftarrow \text{KDF}(k' \| H_2(c))$
- 6: **else**
- 7: $k \leftarrow \text{KDF}(z \| H_2(c))$

Table 1: Parameters of KYBER

Parameter	KYBER512	KYBER768	KYBER1024
κ	2	3	4
η	256	256	256
ρ	3329	3329	3329
ϵ_1	3	2	2
ϵ_2	2	2	2
δ_u	10	10	11
δ_v	4	4	5

$$a[x] \cdot b[x] \pmod{(x^\eta + 1)} = \text{INTT}(\text{NTT}(a) \circ \text{NTT}(b)). \quad (1)$$

The NTT and the INTT both consist of $\log_2(\eta) - 1 = 7$ layers that each contains $\eta/2 = 128$ butterfly operations $\text{Butterfly} : \mathbb{Z}_\rho^2 \rightarrow \mathbb{Z}_\rho^2$. The INTT is typically implemented using the *Gentleman–Sande* (GS) butterfly in Eq. (2), where *twiddle factor* τ is a power of the root of unity ζ . Barrett and Montgomery reduction methods enable efficient and time-constant implementations of modular arithmetic in the prime field \mathbb{Z}_ρ .

$$\text{GSButterfly}(a, b; \tau) \triangleq (a + b, (a - b)\tau) \pmod{\rho}. \quad (2)$$

The lossy compression is defined in Eq. (3). The compression of message coefficients $m \in \mathbb{Z}_\rho$ in Line 6 of Algorithm 3 uses $\delta = 1$, and boils down to Eq. (4).

$$\text{Compress}(x; \rho, \delta) \triangleq \lceil 2^\delta x / \rho \rceil \pmod{2^\delta}, \quad \text{Decompress}(x; \rho, \delta) \triangleq \lceil \rho x / 2^\delta \rceil. \quad (3)$$

$$\text{Compress}(x; \rho, 1) = \begin{cases} 1 & \text{if } \rho/4 < x < 3\rho/4, \\ 0 & \text{otherwise.} \end{cases} \quad (4)$$

The decryption faces an accumulated error on the uncompressed message m' as given in Eq. (5), where summands $\Delta \mathbf{u}$ and Δv denote contributions from the lossy ciphertext compression. Roughly speaking, the compressed message $m \in \{0, 1\}^\eta$ is correct if and only if it holds for each coefficient $i \in [0, \eta - 1]$ that $-\rho/2 < \Delta m_i < \rho/2$.

$$\Delta m = \mathbf{e}^\top \mathbf{r} - \mathbf{s}^\top (\mathbf{e}_1 + \Delta \mathbf{u}) + e_2 + \Delta v \pmod{\pm \rho}. \quad (5)$$

2.2 Key-encapsulation mechanism

KYBER [ABD⁺20] uses a variation of the FO transform that is specified in Algorithms 4 to 6. Essentially, the ciphertext c received by the decapsulation is re-encrypted after decryption and the result c' is compared to c . If this comparison fails, the decapsulation returns a pseudorandom value instead of a failure symbol \perp , which is referred to as *implicit rejection*. Hash functions H_1 and H_2 are instantiated with SHA3-512 and SHA3-256 respectively; the *key-derivation function* (KDF) is instantiated with SHAKE-256. KYBER has a 90s variant with other symmetric primitives, which we do not use.

2.3 ARM Cortex-M4

Following a recommendation by NIST, the ARM Cortex-M4 is the primary *reduced instruction set computer* (RISC) processor for benchmarking the implementation efficiency of PQC schemes. This embedded processor features thirteen 32-bit registers for general purposes, which may pack two 16-bit signed integers. Instructions that perform multiplications, subtractions, and other operations on these halfwords are supported.

Source code for KYBER is publicly available in the *pqm4* library [KRSS]. Although the implementation is largely written in C, we analyze routines written in assembly exclusively. These routines were updated after our analysis, yet similar conclusions can be drawn from the latest version. Given that prime $\rho = 3329 < 2^{12}$, 16-bit halfwords can efficiently store polynomial coefficients whilst providing a margin for lazy reductions, i.e., reductions after additions and subtractions that do not cause overflow may be skipped. As pointed out by Alkim et al. [ABCG20, Algorithm 11], Montgomery reductions can be implemented using two instructions only. The realization from the *pqm4* library is shown in Algorithm 7. The NTT and INTT exclusively rely on these Montgomery reductions, as evidenced by the double GS butterfly in Algorithm 8. Unfortunately, the Montgomery-reduced coefficients lie in the interval $[-\rho + 1, \rho - 1]$ instead of $[0, \rho - 1]$. To obtain coefficients in the interval $[0, \rho - 1]$ right before compression, a slower Barrett reduction is used.

Algorithm 7 Montgomery [KRSS]**Input:** Integer a where $-(\beta/2) \cdot \rho \leq a < (\beta/2) \cdot \rho$ and $\beta = 2^{16}$ **Input:** Prime $\rho = 3329$ **Input:** Negated inverted prime $-\rho^{-1}[31 : 16] = 3327$ **Output:** Reduced $t[31 : 16]$ where $t[31 : 16] = \beta^{-1}a \pmod{\rho}$ and $-\rho < t[31 : 16] < \rho$

- 1: `smulbt` $t, a, -\rho^{-1}$ $\triangleright t \leftarrow (a \pmod{\beta}) \cdot (-\rho^{-1})$
- 2: `smlabb` t, ρ, t, a $\triangleright t[31 : 16] \leftarrow \lfloor ((t \pmod{\beta})\rho + a)/2^{16} \rfloor$

Algorithm 8 DoubleGSButterfly [KRSS]**Input:** $(a[15 : 0], b[15 : 0])$ to first butterfly**Input:** $(a[31 : 16], b[31 : 16])$ to second butterfly**Input:** Twiddle factor $\tau[15 : 0]$ or $\tau[31 : 16]$ **Output:** $(a[15 : 0], b[15 : 0])$ from first butterfly**Output:** $(a[31 : 16], b[31 : 16])$ from second butterfly

- 1: `usub16` t_1, a, b $\triangleright \begin{cases} t_1[15 : 0] \leftarrow a[15 : 0] - b[15 : 0], \\ t_1[31 : 16] \leftarrow a[31 : 16] - b[31 : 16] \end{cases}$
- 2: `uadd16` a, a, b $\triangleright \begin{cases} a[15 : 0] \leftarrow a[15 : 0] + b[15 : 0], \\ a[31 : 16] \leftarrow a[31 : 16] + b[31 : 16] \end{cases}$
- 3: `smulbt/smulbb` b, t_1, τ $\triangleright b \leftarrow t_1[15 : 0] \cdot \tau[\dots]$
- 4: `smultt/smultb` t_1, t_1, τ $\triangleright t_1 \leftarrow t_1[31 : 16] \cdot \tau[\dots]$
- 5: `montgomery` $\rho, -\rho^{-1}, b, t_2$ \triangleright Algorithm 7: reduce b to $t_2[31 : 16]$
- 6: `montgomery` $\rho, -\rho^{-1}, t_1, b$ \triangleright Algorithm 7: reduce t_1 to $b[31 : 16]$
- 7: `pkhtb` $b, b, t_2, \text{asr}\#16$ $\triangleright b[15 : 0] \leftarrow t_2[31 : 16]$

3 Side-Channel Analysis

As specified in Algorithm 6, the decryption is the only building block of KYBER’s decapsulation that uses the private key \hat{s} and is thus the obvious target for SCA. However, SCA-assisted CCAs proposed by D’Anvers et al. [DTVV19], Ravi et al. [RRCB20], and Ueno et al. [UXT⁺21] subverted this intuition. An attacker can construct ciphertexts c such that the correctness of exactly one decrypted message bit $m_i \in \{0, 1\}$ depends on \hat{s} . To avoid the realization of a message-checking oracle through SCA, algorithms that process m should be protected. This includes the hash function H_1 and the entire re-encryption. The academically preferred way of countering SCA is to randomize computations such that dependencies between internal secrets and measurable emissions are weakened. Below, we distinguish between masking methods, which are expensive and substantiated by a security proof in a probing model, and blinding methods, which are cheap and unsupported by a security proof.

3.1 Masking

In masked implementations, finite-ring elements $x \in \mathcal{X}$ are randomly and uniformly split into $\lambda \geq 2$ shares according to Definition 1. According to Lemma 1, one way to meet Definition 1 is to first select $(x^{(2)}, x^{(3)}, \dots, x^{(\lambda)})$ uniformly at random from $\mathcal{X}^{\lambda-1}$, followed by a computation $x^{(1)} = x - x^{(2)} - x^{(3)} - \dots - x^{(\lambda)}$.

Definition 1 (Uniformity). A finite-ring element $x \in \mathcal{X}$ is randomly and uniformly split into $\lambda \geq 2$ shares if $\Pr(x^{(1)}, x^{(2)}, \dots, x^{(\lambda)} \mid x)$ equals $1/|\mathcal{X}|^{\lambda-1}$ if $x^{(1)} + x^{(2)} + \dots + x^{(\lambda)} = x$ and 0 otherwise.

Lemma 1 (Subset of Shares). *For a finite-ring element $x \in \mathcal{X}$ that is randomly and uniformly split into λ shares according to Definition 1, any tuple of $\lambda-1$ shares is uniformly distributed on $\mathcal{X}^{\lambda-1}$ and thus independent of x . More generally, any tuple of $\alpha \in [1, \lambda-1]$ shares is uniformly distributed on \mathcal{X}^α .*

We distinguish between (i) Boolean masking, where $\mathcal{X} = \{0, 1\}^\sigma$ and additions are defined by XORing, and (ii) arithmetic masking, where $x \in \mathbb{Z}_\rho$, and additions are performed modulo a prime ρ . For efficiency reasons, Boolean masking is typically used for symmetric-key algorithms, whereas arithmetic masking is used for polynomial operations. Hence, Boolean-to-arithmetic and arithmetic-to-Boolean conversions are commonplace.

A function $G : \mathcal{X} \rightarrow \mathcal{Y}$ must also be split such that shares of $x \in \mathcal{X}$ satisfying Definition 1 are mapped to shares of $y = G(x)$ that again satisfy Definition 1. If G is linear, G is trivially split by defining $\forall i \in [1, \lambda] : G^{(i)}(x^{(i)}) \triangleq G(x^{(i)})$, considering that $G^{(1)}(x^{(1)}) + G^{(2)}(x^{(2)}) + \dots + G^{(\lambda)}(x^{(\lambda)}) = G(x^{(1)} + x^{(2)} + \dots + x^{(\lambda)}) = G(x)$. For lattice-based cryptography, linear components include polynomial additions, the NTT, and the INTT. Non-linear components, such as **Compress** in Eq. (3) and the polynomial comparison, require custom-developed masking schemes [BGR⁺21].

3.2 Blinding

For blinding methods, we distinguish between randomization of data and randomization of time. The latter can be achieved by randomly permuting the order of parallelizable operations [Saa18, OSPG18, RPBC20, PP21a]. For example, the polynomial coefficients fed into **Compress** and **Decompress** in Eq. (3) can be permuted. Similarly, the butterfly operations within an NTT/INTT layer can be shuffled.

As an example of data randomization, consider a finite-field multiplication $y = G(x_1, x_2) = x_1 \cdot x_2$, where $x_1, x_2, y \in \mathbb{F}$. Saarinen [Saa18] proposed computing $(r_1 \cdot r_2)^{-1} \cdot G(x_1 \cdot r_1, x_2 \cdot r_2)$ where r_1 and r_2 are chosen randomly, uniformly, and independently from $\mathbb{F} \setminus \{0\}$. For a prime-field multiplication $y = G(x, \zeta^i)$ within an NTT/INTT, where $x, y \in \mathbb{Z}_\rho$ and $i \in [0, \eta - 1]$, costs can be lowered by computing $\zeta^{-r_1-r_2} \cdot G(x \cdot \zeta^{r_1}, \zeta^{i+r_2})$ where r_1 and r_2 are chosen randomly, uniformly, and independently from $[0, \eta - 1]$. At least, if a lookup table of the powers of ζ is available. Ravi et al. [RPBC20] applied the latter technique at various granularities: the extent to which r_1 and r_2 are reused across the multiplications within an NTT/INTT layer is a trade-off between cost and security. In its most generic form, the GS butterfly in Eq. (2) is realized as in Eq. (6), where blinding factors ζ^{r_1} and ζ^{r_2} cancel out to a factor 1 after the last layer.

$$\text{BlindedGSButterfly}(a, b; \zeta^i) \triangleq ((a + b) \zeta^{r_1}, (a - b) \zeta^{i+r_2}) \pmod{\rho}. \quad (6)$$

4 Fault Attacks

Although fault attacks on the key generation and the encapsulation exist [VOGR18, RRB⁺19], the decapsulation is once again particularly vulnerable. An attacker can fault this module a virtually unlimited number of times in order to retrieve the private key s .

4.1 Differential fault analysis

As pointed out by Oder et al. [OSPG18], a positive side effect of using the FO transform is that many fault attacks on the decapsulation are inherently countered: by re-encrypting the decrypted message m and comparing the result c' to the externally provided ciphertext c , secret-revealing faulted data is kept internal instead of forwarded to the output. This countermeasure, which also exists in a simpler form where an encryption or decryption is executed twice, is well-established since the early 2000s, at which time Karri et

al. [KWMK02] protected *block ciphers* such as the *Advanced Encryption Standard* (AES) against *differential fault analysis* (DFA). For block ciphers, the countermeasure can only be defeated through a double fault injection: a fault in the encryption can compensate a fault in the decryption such that the equality-check is passed, or a fault can skip the equality-check so that an arbitrary fault in the encryption propagates to the output. Unfortunately, and as surveyed by Xagawa et al. [XIU⁺21], the lattice-based version can be broken through a single fault that skips the equality check, considering that CCA resistance is removed this way.

4.2 Ineffective Faults

Another concern is that the inherent FO defense only counters DFA, or more generally, any attack that leverages faulted data. As already established in the 2000s, mere knowledge of whether the output of a keyed cryptographic algorithm is wrong or correct after introducing a fault might leak information about the secret. Faults that have the latter effect are often referred to as *safe errors* [YJ00] or *ineffective faults* [Cla07]. Because the attacker gains at most one bit of information per injected fault, a full key recovery typically requires more injections than with DFA. Bettale, Montoya, and Renault [BMR21] proposed an ineffective-fault attack against several lattice-based schemes, but KYBER is deemed secure. Attacks presented at CHES 2021 [PP21a] and IndoCrypt 2021 [HPP21a], the latter of which is based on the former, target KYBER and are recapitulated below.

4.2.1 CHES 2021

Pessl and Prokop [PP21a] skipped a subtraction instruction in the final compression step of KYBER’s decryption, i.e., $m \leftarrow \text{Compress}(m'; \rho, 1)$ in Line 6 in Algorithm 3, such that m' and thus also the accumulated error Δm in Eq. (5) increases by $\lfloor \rho/4 \rfloor$. This increase is represented in Eq. (7), where index $\iota \in [0, \eta - 1]$ points to the targeted polynomial coefficient, and also in Fig. 1 through a quarter counterclockwise turn. Values of m'_ι and $(m'_\iota)^*$ that compress to $m = 0$ and $m = 1$ are colored orange and blue respectively. The radial indents represent a *probability mass function* (PMF), although not drawn to scale for visibility. Recall that a single erroneous message bit $m'_\iota \neq m_\iota$ triggers an avalanche of errors in coins $r = H_1(m \| h)$ and thus also in the re-encryption result $c' = \text{Encrypt}(k_{\text{pub}}, m, r)$. Roughly speaking, the effectiveness of the fault reveals the sign of the accumulated error Δm_ι . The exact version is given in Eq. (8), which takes into account that the real numbers $\rho/4$, $\rho/2$, and $3\rho/4$ do not coincide with integer representatives in \mathbb{Z}_ρ .

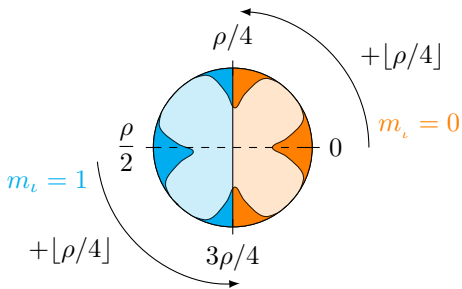


Figure 1: Rotation of m'_ι .

$$(m'_\iota)^* = m'_\iota + \lfloor \rho/4 \rfloor \implies \Delta m'_\iota = \Delta m_\iota + \lfloor \rho/4 \rfloor. \quad (7)$$

$$m'_\iota \neq m_\iota \iff \Delta m_\iota \geq t_\iota, \quad (8a)$$

$$\text{where } t_\iota \triangleq \begin{cases} 1 & \text{if } m_\iota = 0, \\ 0 & \text{if } m_\iota = 1. \end{cases} \quad (8b)$$

From Eqs. (5) and (8), it follows that the observed effectiveness of the fault provides one inequality that is linear in the secret $\mathbf{x} \triangleq (\mathbf{s}, \mathbf{e}) \in [-\epsilon_1, \epsilon_1]^\psi$ where $\psi \triangleq 2\kappa\eta$, as given in Eq. (9). Both \mathbf{a} and \mathbf{b} are entirely determined by the encapsulation and are thus not

only known but also controllable by the attacker. Reductions modulo prime ρ are omitted because (i) b and the elements of \mathbf{a} and \mathbf{x} are small in absolute value, and (ii) opposite signs ensure that $\mathbf{a}\mathbf{x} + b$ is small in absolute value too. Thanks to this omission, and by gathering ω inequalities where ω is several thousands, the system comprising a matrix \mathbf{A} with size $\omega \times \psi$ and a vector \mathbf{b} of length ω can be solved for the secret \mathbf{x} . In practice, the index ι is kept the same for all ω inequalities so that the fault-injection setup only needs to be aligned with a single point in space and time. From Eq. (9b) and the coin-based randomization of the encapsulation, it follows that making ι constant and selecting ι uniformly at random from $[0, \eta - 1]$ for each inequality would result in equally solvable systems anyway.

$$\mathbf{a}\mathbf{x} + b \begin{cases} \geq 0 & \text{if decapsulation fails,} \\ < 0 & \text{otherwise,} \end{cases} \quad \mathbf{x} \triangleq \begin{pmatrix} \text{Poly2Vec}(\mathbf{s}[0]) \\ \vdots \\ \text{Poly2Vec}(\mathbf{s}[\kappa - 1]) \\ \text{Poly2Vec}(\mathbf{e}[0]) \\ \vdots \\ \text{Poly2Vec}(\mathbf{e}[\kappa - 1]) \end{pmatrix}, \quad \text{Poly2Vec}(p) \triangleq \begin{pmatrix} p_0 \\ \vdots \\ p_{\eta-1} \end{pmatrix}, \quad (9a)$$

where $b \triangleq (e_2 + \Delta v - t)_\iota$,

$$\mathbf{a}\boldsymbol{\tau} \triangleq \begin{pmatrix} -\text{Poly2VecX}(\mathbf{e}_1[0] + \Delta\mathbf{u}[0]; \iota) \\ \vdots \\ -\text{Poly2VecX}(\mathbf{e}_1[\kappa - 1] + \Delta\mathbf{u}[\kappa - 1]; \iota) \\ \text{Poly2VecX}(\mathbf{r}[0]; \iota) \\ \vdots \\ \text{Poly2VecX}(\mathbf{r}[\kappa - 1]; \iota) \end{pmatrix}, \quad \text{Poly2VecX}(p; \iota) \triangleq \begin{pmatrix} p_\iota \\ \vdots \\ p_0 \\ -p_{\eta-1} \\ \vdots \\ -p_{\iota+1} \end{pmatrix}. \quad (9b)$$

In practice, the obtained inequalities might be incorrect. Void injections where no fault is actually introduced can be misclassified as an ineffective fault. Similarly, injections that cause faults other than the intended instruction skip can be misclassified as an effective fault: the returned symmetric key is $k \leftarrow \text{KDF}(z \| \text{H}_2(c))$ either way. In order to tolerate errors, and in order to keep execution times reasonable despite the large dimensions (ω, ψ) , the authors abstain from using *linear programming* and base their solver on belief propagation. Their eventual algorithm, however, was unable to exceed a 1% error rate, and attempts to increase this number were deferred to *future work*.

The solver is an iterative method that maintains a PMF for each unknown $\mathbf{x}[j]$ in Eq. (9), where $j \in [0, \psi - 1]$. All ψ PMFs are initialized with the CBD on $[-\epsilon_1, \epsilon_1]$, and in each iteration, all ψ PMFs are updated, until they sufficiently approximate one-point distributions on $[-\epsilon_1, \epsilon_1]$ to make further iterations pointless. To update the PMF of any given $\mathbf{x}[j]$, the probability that $\mathbf{a}\mathbf{x} + b \geq 0$ in Eq. (9) holds is computed for each possible outcome of $\mathbf{x}[j] \in [-\epsilon_1, \epsilon_1]$ and for each out of ω inequalities according to Eq. (10), and these $2\epsilon_1\omega$ probabilities are then aggregated. Importantly, the PMF updates do not interfere with one another, i.e., each update only uses probability masses from the previous iteration.

$$\begin{aligned} \forall i \in [0, \omega - 1], \\ \forall j \in [0, \psi - 1], \quad \mathbf{P}[i, j, k] &= \Pr \left(\mathbf{A}[i, j] (k - \epsilon_1) + \left(\sum_{j' \in [0, \psi - 1] \setminus \{j\}} \mathbf{A}[i, j'] \mathbf{x}[j'] \right) + \mathbf{b}[i] \geq 0 \right). \\ \forall k \in [0, 2\epsilon_1], \end{aligned} \quad (10)$$

Each probability in Eq. (10) involves a linear combination of $\psi - 1$ random variables $\mathbf{x}[j]$, which do not have a special shape anymore after the first iteration, and generally necessitates linear (non-circular) convolution. Even though the *fast Fourier transform*

(FFT) is used to accelerate these convolutions, similar to how the NTT is used in KYBER to accelerate polynomial multiplication according to Eq. (1), and even though *binary trees* improve the reuse of intermediate variables, the computational load remains heavy with $\omega\psi$ FFTs and $\omega\psi$ inverse FFTs per iteration.

4.2.2 Indocrypt 2021

Hermelink, Pessl, and Pöppelmann [HPP21a] presented a similar attack: Eqs. (7) to (10) and Fig. 1 remain identical. The difference is that the increase of m'_i by $\lfloor \rho/4 \rfloor$ in Eq. (7) is not realized by a fault but by manipulating the compressed ciphertext coefficient v_i of an otherwise correctly computed encapsulation as specified in Eq. (11) and enabled by $m' \leftarrow v' - \text{INTT}(\hat{\mathbf{s}}^\top \circ \hat{\mathbf{u}})$ in Line 5 in Algorithm 3. Similar manipulations were used in the aforementioned SCA-assisted CCAs [DTVV19, RRCB20, UXT⁺21], with the difference that the message-checking oracle is now realized by reintroducing a fault instead of SCA. To authors make the decapsulation succeed in case of a correctly decrypted message bit $m_i^* = m_i$ by the suggested use of a laser to flip a single bit in either input operand of the ciphertext comparison. If multiple bits can reliably be flipped, the *Hamming distance* (HD) constraint can be removed. Depending on which operand is faulted, the symmetric key k is either $\text{KDF}(k' \| \text{H}_2(c))$ or $\text{KDF}(k' \| \text{H}_2(c^*))$; both values are known to the attacker. If $m_i^* \neq m_i$, the coins r used by the re-encryption and thus also the produced ciphertext c' are entirely corrupted, and any attempt for rectification is in vain: $k \leftarrow \text{KDF}(z \| \text{H}_2(c^*))$.

$$(v'_i)^* = v'_i + \lfloor \rho/4 \rfloor \text{ and } v_i^* = \text{Compress}((v'_i)^*), \quad \text{where } \text{HD}(v_i, v_i^*) = 1. \quad (11)$$

Unlike the attack at CHES 2021, the error rate of the obtained inequalities is inherently asymmetric: an observed decapsulation success cannot be misclassified, whereas an observed decapsulation failure can be misclassified due to void injections and unintended faults. To reduce the error rate in the latter case, $\beta > 1$ fault-injection attempts per inequality can be made. The solver is another variation of belief propagation making use of Eq. (10), FFTs, and binary trees, and is fed inequalities that are 100% correct, originating from perfect software-simulated faults. Around 6000, 7000, and 9000 faulted decapsulations suffice to recover the private key of KYBER512, KYBER768, and KYBER1024 respectively, with a success rate of nearly 100%. To achieve an execution time under 10 minutes for Kyber768 with 7000 inequalities, 32 threads running on 16 cores are required.

The attack may be hindered by masking, shuffling, and/or double executions, but is not precluded, in part due to the error-rate asymmetry. Therefore, the authors proposed an additional countermeasure: instead of ciphertexts c , pairs $(c, \text{H}_3(c))$ where H_3 is another hash function are stored in *random-access memory* (RAM) and eventually compared. Although faulting c while it is stored in RAM becomes pointless, the attack still succeeds by faulting c before it is fed into the hash function, e.g., in the back end of $\text{Compress}(v; \rho, \delta_v)$.

5 Roulette Attacks

Considering that our roulette attacks may be applicable to several KEMs, we first present a general methodology in Section 5.1, and then apply this methodology to KYBER's decapsulation in Section 5.2.

5.1 General Methodology

Consider a keyed cryptographic algorithm $A : \mathcal{S} \times \mathcal{I} \rightarrow \mathcal{O}$ where $s \in \mathcal{S}$ is keying material, $i \in \mathcal{I}$ is the public input, and $o \in \mathcal{O}$ is the output. Output o is not necessarily public, but an attacker can observe whether or not o is correct. We decompose A into four parts, as

shown in Fig. 2. To keep the execution time of the attack within bounds, we require that the cardinalities $|\mathcal{T}_1|$, $|\mathcal{T}_2|$, and $|\mathcal{T}_3|$ are small.

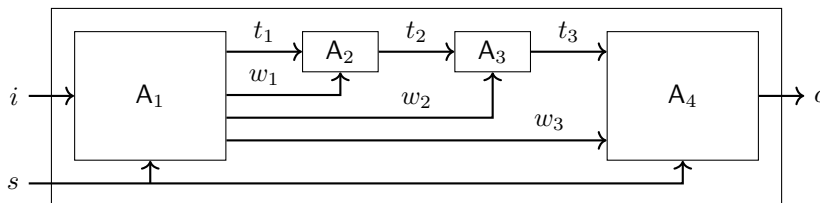


Figure 2: Decomposition of cryptographic algorithm A.

For a constant input (s, i) , the attacker repeatedly faults either t_1 or A_2 or t_2 or A_3 or t_3 such that $t_3^* \in \mathcal{T}_3$ is not constant, i.e., t_3^* does not follow a one-point distribution with respect to the infinite set of fault injections. If for the given distribution of t_3^* , the probability that A fails to produce the correct output o depends on the secret $s \in \mathcal{S}$, then the attacker retrieves information on s . Although many distributions might enable an attack, we idealize the case where t_3^* is uniformly distributed on \mathcal{T}_3 . In our casino analogy, this corresponds to spinning a fair roulette wheel, at least if we visualize \mathcal{T}_3 through a circular representation. Our motivation for this idealization is that uniform distributions naturally support (i) a large attack surface, as shown by the fault-propagation properties in Section 5.1.1, and (ii) various fault models, as shown by the examples in Section 5.1.2.

5.1.1 Attack Surface

For a function that is *balanced* according to Definition 2, which extends existing definitions [DMB19], uniformly distributed faults propagate as uniformly distributed faults, as formalized in Lemma 2 and proven in Appendix C.1. If the function $A_3 : \mathcal{T}_2 \times \mathcal{W}_2 \rightarrow \mathcal{T}_3$ in Fig. 2 happens to be balanced with respect to t_2 , an attacker who is able to fault either A_2 or t_2 such that the faulted value $t_2^* \sim U(\mathcal{T}_2)$, indirectly achieves $t_3^* \sim U(\mathcal{T}_3)$. If, additionally, A_2 is balanced with respect to t_1 , then a faulted value $t_1^* \sim U(\mathcal{T}_1)$ has the same effect.

Definition 2 (Balanced Function). Let $G : \mathcal{X} \rightarrow \mathcal{Y}$ be a function. If it holds $\forall y \in \mathcal{Y}$ that $|\{x \in \mathcal{X} \mid G(x) = y\}| = |\mathcal{X}|/|\mathcal{Y}|$, then G is balanced. Similarly, for a function $G : \mathcal{X}_1 \times \mathcal{X}_2 \rightarrow \mathcal{Y}$, if it holds $\forall (x_2, y) \in \mathcal{X}_2 \times \mathcal{Y}$ that $|\{x_1 \in \mathcal{X}_1 \mid G(x_1, x_2) = y\}| = |\mathcal{X}_1|/|\mathcal{Y}|$, then G is balanced with respect to input $x_1 \in \mathcal{X}_1$.

Lemma 2 (Fault Propagation by Balanced Functions). *Let $G : \mathcal{X} \rightarrow \mathcal{Y}$ be a balanced function, as formalized in Definition 2. If $x \sim U(\mathcal{X})$, then $y \sim U(\mathcal{Y})$. Similarly, for a function $G : \mathcal{X}_1 \times \mathcal{X}_2 \rightarrow \mathcal{Y}$ that is balanced with respect to input $x_1 \in \mathcal{X}_1$, if $x_1 \sim U(\mathcal{X}_1)$ is independent of $x_2 \in \mathcal{X}_2$, then $y \sim U(\mathcal{Y})$.*

Fortunately for the attacker, balanced functions are frequently used in cryptography. Bijections are a trivial example. Additions in a finite ring and multiplications in a finite field are two more examples, as formalized in Lemmas 3 and 4 respectively, and proven in Appendices C.2 and C.3 respectively. In fact, balancedness is merely the ideal case; imbalanced fault propagation might still enable an attack in practice.

Lemma 3 (Balancedness of Addition in Finite Rings). *Let \mathcal{R} be a finite ring and let $G : \mathcal{R}^2 \rightarrow \mathcal{R}$ be defined as $y \triangleq G(x_1, x_2) \triangleq x_1 + x_2$. It holds that G is fully balanced, i.e., Definition 2 is met with respect to both input $x_1 \in \mathcal{R}$ and $x_2 \in \mathcal{R}$.*

Lemma 4 (Balancedness of Multiplication in Finite Fields). *Let \mathcal{F} be a finite field and let $G : \mathcal{F}^2 \rightarrow \mathcal{F}$ be defined as $y \triangleq G(x_1, x_2) \triangleq x_1 \cdot x_2$, where $x_2 \neq 0$. It holds that G is balanced, i.e., Definition 2 is met with respect to input $x_1 \in \mathcal{F}$.*

5.1.2 Fault Models

Examples 1 to 4 demonstrate that the ideal distribution, $t_3^* \sim U(\mathcal{T}_3)$, can be achieved for various fault models. Masking is a facilitator in Examples 2 and 3; data-randomizing blinding is a facilitator in Example 4. Depending on whether A_2 and A_3 are balanced, Examples 1 and 2 can be applied to t_1 , t_2 , and/or t_3 , and Examples 3 and 4 can be applied to A_2 and/or A_3 . At least in the ideal case, because non-uniform distributions also enable attacks in practice.

Example 1 (Random Faults). Random faults where $y^* \sim U(\mathcal{Y})$ comprise a well-established fault model in the academic literature and are covered by definition. Also stronger fault models where $y \in \{0, 1\}^\varphi$ is XORed with an attacker-chosen error $e \in \{0, 1\}^\varphi$ are covered. If the attacker chooses $e \sim U(\{0, 1\}^\varphi)$, then $y^* \triangleq y \oplus e \sim U(\{0, 1\}^\varphi)$.

Example 2 (Set-To-Constant Faults). Set-to-0 and set-to-1 faults are covered for masked implementations. Let y be randomly and uniformly split into $\lambda \geq 2$ shares according to Definition 1, and without loss of generality, assume that the first share, $y^{(1)} \in \mathcal{Y}$, is set to an arbitrary constant $\theta \in \mathcal{Y}$, whereas shares $y^{(2)}, \dots, y^{(\lambda)} \in \mathcal{Y}$ are untouched. Considering that $y^{(1)} \sim U(\mathcal{Y})$ and $(y^{(2)}, \dots, y^{(\lambda)}) \sim U(\mathcal{Y}^{\lambda-1})$ according to Lemma 1, it follows that the faulted value $y^* = \theta + y^{(2)} + \dots + y^{(\lambda)} = y - y^{(1)} + \theta \sim U(\mathcal{Y})$.

Example 3 (Instruction Skips and Corruptions). Let $G : \mathcal{X} \rightarrow \mathcal{Y}$ be realized through a masked software implementation. Without loss of generality, assume that an instruction in the first share function, $G^{(1)}$, is either skipped or corrupted such that the faulty output share $(y^{(1)})^*$ is independent of the correct output share $y^{(1)}$. Hence, $y^* = (y^{(1)})^* + y^{(2)} + \dots + y^{(\lambda)}$ is again uniformly distributed on \mathcal{Y} .

Example 4 (Arbitrary Bit Flips). Let $G : \mathcal{X} \rightarrow \mathcal{Y}$ be an affine function over a finite field $\mathcal{X} = \mathcal{Y} = \{0, 1\}^\varphi$ where addition is defined by XORing. Let $y \triangleq G(x)$ be realized through a blinded implementation $y = r^{-1} G(r \cdot x)$ where $r \sim U(\{0, 1\}^\varphi \setminus \{0\})$. For any pattern of bit flips $e \in \{0, 1\}^\varphi \setminus \{0\}$ applied to the input of G , it holds that the faulted output $y^* \triangleq r^{-1} G(r \cdot x \oplus e) = y \oplus r^{-1} G(e) \sim U(\{0, 1\}^\varphi \setminus \{y\})$. Strictly speaking, this distribution is nearly uniform, given that the case $y^* = y$ is excluded. One could achieve $y^* \sim U(\{0, 1\}^\varphi)$ by aborting the fault injection with probability $1/2^\varphi$, but this would be pointless in an actual attack.

5.1.3 Comparisons

Table 2 compares our roulette attacks to well-known fault attacks, i.e., DFA, *fault sensitivity analysis* (FSA) [LOS12], and a *statistical ineffective fault attack* (SIFA) [DEK⁺18]. The standout property of roulette attacks is that masking is a facilitator. Although masking may not preclude DFA [BH08], FSA [MMP⁺11, Del20], or SIFA [DEG⁺18], it is not a facilitator here. Furthermore, note that the fault distributions of roulette attacks and SIFA are complementary to some extent.

5.2 Application to Kyber’s Decapsulation

We now instantiate the generic cryptographic algorithm A from Section 5.1 with KYBER’s decapsulation, as specified in Algorithm 6. Our first and foremost roulette attack is an extension of the IndoCrypt attack [HPP21a]; the private key s is recovered by faulting the re-encryption. A second roulette attack recovers the message m and the corresponding session key k by faulting the decryption. Considering that the second attack is far less practical while recovering the short-term and thus not the long-term secret, its specification is deferred to Appendix B.

Table 2: Comparison of fault attacks.

Technique	DFA	FSA	SIFA	Roulette
Input i	Unknown	Known	Unknown	Known
Correct output o	Known	Unknown	Known	Unknown
Faulty output o^*	Known	Unknown	Unknown	Unknown
Input i	Constant i	Constant i	$i \leftarrow U(\mathcal{I})$	Constant i
Correct intermediate t	Constant t	Constant t	$t \sim U(\mathcal{T})$	Constant t
Faulty intermediate t^*	Any	Any	$t^* \not\sim U(\mathcal{T})$	Any, $t^* \sim U(\mathcal{T})$
Fault intensity	Constant	Variable	Constant	Constant
Masking	Nuisance	Nuisance	Nuisance	Facilitator
Duplication	Game over	Don't care	Don't care	Nuisance

5.2.1 Attack Surface

The generic variable $t_3 \in \mathcal{T}_3$ in Fig. 2 is instantiated with a compressed ciphertext coefficient $v_l \in \{0, 1\}^{\delta_v}$ that is output from the re-encryption, as specified in Algorithm 2. Following Hermelink et al. [HPP21a], the goal is to match a manipulated coefficient so that the polynomial comparison succeeds, at least if the preceding decryption is correct. If the faulted value v_l^* is uniformly distributed on $\{0, 1\}^{\delta_v}$, then the probability of a successful decapsulation is approximately 0 if $m_l \neq m_l^*$ and $1/2^{\delta_v}$ otherwise. For KYBER512 and KYBER768, the latter probability is $1/16$; for KYBER1024, the latter probability is $1/32$. The attacker injects faults until a decapsulation success is observed. After β unsuccessful injections, a decapsulation failure is assumed. Inequalities that correspond to an observed decapsulation success are always correct, whereas the error rate of inequalities that correspond to an observed decapsulation failure decreases with β .

Compared to the Indocrypt attack [HPP21a] in its original form, the number of fault injections increases by roughly one or two orders of magnitude, but we get a considerably larger attack surface and support for various fault models in return. As illustrated in Fig. 3 and in accordance with the C reference implementation of KYBER submitted to NIST [ABD⁺20], the function $A_3 \circ A_2$ that produces a coefficient $v_l \in \{0, 1\}^{\delta_v}$ comprises (i) one GS butterfly in the last layer of the INTT, which includes one Montgomery multiplication, (ii) another Montgomery multiplication for scaling purposes, (iii) the generation of one CBD sample, (iv) the decompression of one message bit, (v) one addition, (vi) one Barrett reduction, and (vii) one compression. Moreover, by faulting any of these building blocks, the countermeasure of Hermelink et al. [HPP21a] to store $(c, H_3(c))$ in RAM is bypassed.

Another godsend for the attacker is that the fault-propagation statistics are almost ideal. The modular addition is perfectly balanced according to Definition 2 with respect to all three inputs (this is a trivial generalization of Lemma 3). Ciphertext compression as defined in Eq. (3) is not perfectly balanced, but the deviation is too small to notably impact the attack. If we introduce faults such that the uncompressed coefficient $(v_l')^*$ is uniformly distributed on $[0, \rho - 1]$, then the compressed coefficient $(v_l)^*$ slightly deviates from uniform. For KYBER512 and KYBER768, the zero coefficient occurs with probability $209/3329$, whereas all other coefficients occur with probability $208/3329$. Similarly, for KYBER1024, this becomes $105/3329$ for the zero coefficient and $104/3329$ for all other coefficients.

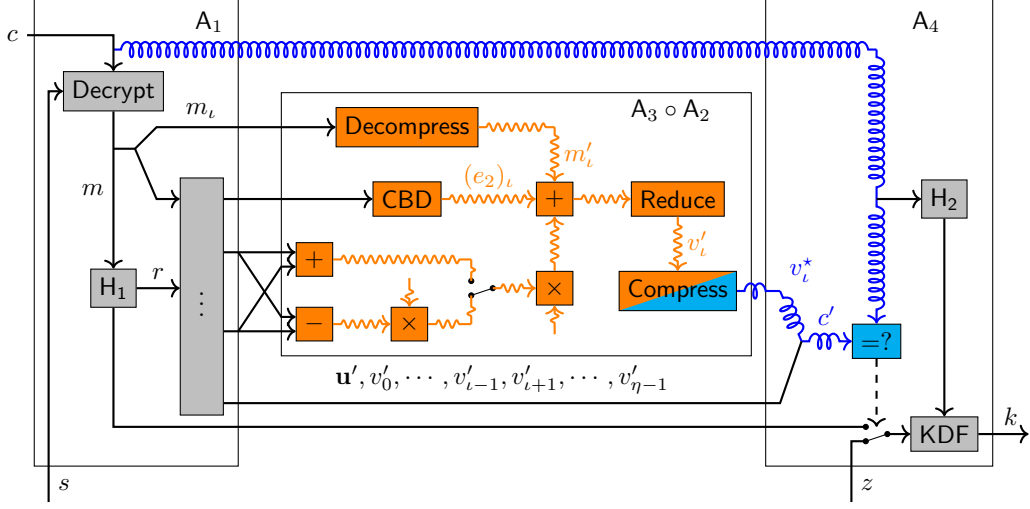


Figure 3: The attack surface of the IndoCrypt paper [HPP21a] is colored blue; our extension is colored orange.

5.2.2 Optional Hamming-Distance Constraint

The sole purpose of the HD constraint in Eq. (11) is to establish single bit flips as the fault model. In our extension of the attack, this constraint does not affect the feasibility of a fault injection and is thus entirely optional. To accommodate a potential omission, we replace Eqs. (8b) and (11). As a starting point, we summarize the behavior of **Compress** and **Decompress** in Eq. (3). For **KYBER512** and **KYBER768**, where $\delta_v = 4$, our summary is contained in the first five columns of Table 3. The first and last elements of each bin are defined by **Compress**; the bin centers are defined by **Decompress**. For brevity, we do not discuss **KYBER1024**, where $\delta_v = 5$, but identical conclusions can be drawn from Table 5 in Appendix A.

An evident anomaly is that bin 0 is ‘oversized’: it contains 209 elements, whereas 15 ‘ordinary’ bins each contain 208 elements. The proposed manipulation in Eq. (11) is to add $\lfloor \rho/4 \rfloor = 832 = 4 \cdot 208$ to the uncompressed coefficient v'_i , which is a jump spanning exactly 4 ‘ordinary’ bins. Unfortunately, the first element of bin 0 then maps to the last element of bin 3, given that $3225 + 832 \bmod 3329 = 728$, and thus not to the first element of bin 4. In absence of the HD constraint, the decryption would face an accumulated error $\Delta m'_i = \Delta m_i + 632$, which significantly undershoots the desired effect $\Delta m'_i = \Delta m_i + 832$ in Eq. (7). An easy fix is to replace Eq. (11) by a direct manipulation of the compressed coefficient v_i as specified in Eq. (12).

$$v_i^* = v_i + 2^{\delta_v - 2} \bmod 2^{\delta_v}. \quad (12)$$

Furthermore, in cases where the HD is 2 instead of 1, the accumulated error Δm_i happens to be increased by 833 instead of 832. Equation (13) extends Eq. (8b) accordingly.

$$t_i \triangleq \begin{cases} 1 & \text{if } m_i = 0 \text{ and } \Delta m_i^* = \Delta m_i + 832, \\ -1 & \text{if } m_i = 1 \text{ and } \Delta m_i^* = \Delta m_i + 833, \\ 0 & \text{otherwise.} \end{cases} \quad (13)$$

5.2.3 Masked Software on ARM Cortex-M4

To demonstrate how roulette attacks can defeat SCA countermeasures, theoretical examples are given. Due to the large attack surface in Fig. 3, where most building blocks come

Table 3: Properties of the compressed ciphertext coefficients $v \in [0, 2^\delta - 1]$ where $\delta = 4$.

Original					Manipulated			
Bin	Size	First	Last	Center	Bin	Fault	HD	Δm^*
0	209	3225	104	0	4			
1		105	312	208	5	0100	1	$\Delta m + 832$
2		313	520	416	6			
3		521	728	624	7			
4		729	936	832	8			
5		937	1144	1040	9	1100	2	$\Delta m + 833$
6		1145	1352	1248	10			
7	208	1353	1560	1456	11			
8		1561	1768	1665	12			
9		1769	1976	1873	13	0100	1	$\Delta m + 832$
10		1977	2184	2081	14			
11		2185	2392	2289	15			
12		2393	2600	2497	0			
13		2601	2808	2705	1	1100	2	$\Delta m + 833$
14		2809	3016	2913	2			
15		3017	3224	3121	3			

with a plethora of implementation strategies and masking schemes, we cannot possibly be exhaustive. Our first example is a segment of masked software on the ARM Cortex-M4. Although the KYBER implementations in the *pqm4* library [KRSS] are unprotected, we focus on linear functions exclusively so that masking is realized merely by executing the corresponding code segments $\lambda \geq 2$ times on their respective shares. More specifically, we focus on linear functions that are written in assembly so that differences among C compilers and build settings are irrelevant. We opted for the double GS butterfly in the last layer of the INTT, as implemented in Algorithm 8 and executed on $\lambda \geq 2$ shares. For all nine instructions, Table 4 summarizes the effect of skipping that particular instruction for a single share.

Table 4: The impact of an instruction skip on the double GS butterfly in Algorithm 8 where one out of $\lambda \geq 2$ shares is targeted. A checkmark (✓) denotes the correct result.

Skipped instruction	c_1^*	d_1^*	c_2^*	d_2^*
1 usub16 t_1, a, b	✓	$\sim U(\mathbb{Z}_\rho)$	✓	$\sim U(\mathbb{Z}_\rho)$
2 uadd16 a, a, b	$\sim U(\mathbb{Z}_\rho)$	✓	$\sim U(\mathbb{Z}_\rho)$	✓
3 smulbb b, t_1, τ	✓	$\sim U(\mathbb{Z}_\rho)$	✓	✓
4 smultb t_1, t_1, τ	✓	✓	✓	$\sim U(\mathbb{Z}_\rho)$
5.1 smulbt $t_2, b, -\rho^{-1}$	✓	$\not\sim U(\mathbb{Z}_\rho)$	✓	✓
5.2 smlabb t_2, ρ, t_2, b	✓	$\not\sim U(\mathbb{Z}_\rho)$	✓	✓
6.1 smulbt $b, t_1, -\rho^{-1}$	✓	✓	✓	$\not\sim U(\mathbb{Z}_\rho)$
6.2 smlabb b, ρ, b, t_1	✓	✓	✓	$\not\sim U(\mathbb{Z}_\rho)$
7 pkhtb $b, b, t_2, \text{asr}\#16$	✓	$\sim U(\mathbb{Z}_\rho)$	✓	✓

Clearly, the attacker is in a privileged position: for five out of nine instruction skips, the faulted output coefficients are uniformly distributed, which is our ideal-case scenario. The uniformity proofs are all instances of Example 3 and deferred to Appendix C.4. For the first two instruction skips though, two output coefficients are disturbed, which implies

that the attacker must perform more fault injections. For instructions 5.1 to 6.2 in Table 4, a tractable closed-form expression for the distribution of the faulted coefficient d^* might not exist. However, we took an empirical approach by measuring the distribution of d^* on the ARM Cortex-M4, where an instruction skip is trivially realized by removing that particular instruction from the source code, and did not observe any non-uniformities that would hinder the attack.

5.2.4 Blinded Hardware

For attacks on hardware components, spatially localized fault-injections methods such as lasers beams or electromagnetic waves are of particular interest. A potential target is, for example, a GS butterfly blinded according to Eq. (6) in the final INTT layer. As formalized in Eq. (14), if the attacker flips an arbitrary set of bits in multiplicand $(a + b)$, then the faulted butterfly output c^* is uniformly distributed on a subset of \mathbb{Z}_ρ with cardinality η , given that ζ is the η -th root of unity. Contrary to Example 4, only $\eta/\rho \approx 7.7\%$ of all possible values are covered, but the attack succeeds considering that one or more values around $\Delta c = \lfloor \rho/4 \rfloor$ suffice.

$$(a + b)^* \triangleq (a + b) \oplus e \implies \Delta c \triangleq c^* - c = \left(\sum_{n=0}^{\lfloor \log_2(\rho) \rfloor} e[n](-1)^{(a+b)[n]}2^n \right) \zeta^i. \quad (14)$$

Similarly, bit flips in multiplicand $(a - b)$ cause butterfly output d to be uniformly distributed on a subset of η elements in \mathbb{Z}_ρ . Flipping bits in either a or b is possible too, but then more injections must be performed because c and d are simultaneously faulted.

5.2.5 Countermeasures

As demonstrated in Sections 5.2.3 and 5.2.4, masking and blinding methods that randomize intermediate variables facilitate roulette attacks. Other off-the-shelf countermeasures slow down the attack, albeit at a significant cost. For example, against a re-encryption module in which polynomial coefficients are randomly permuted, the attacker must inject faults until the chosen time of the injection eventually coincides with the manipulated ciphertext coefficient v_i^* . Alternatively, if the re-encryption and ciphertext comparison are executed twice, the attacker needs two consecutive lucky spins of the roulette wheel.

6 Solving Systems of Linear Inequalities

Both Pessl and Prokop [PP21b] and Hermelink et al. [HPP21b] published source code for solving systems of linear inequalities on GitHub, but we implement our own solver from scratch in order to reduce the computation time and increase the error tolerance. Source code is available in the following GitHub repository: <https://github.com/Crypto-TII/roulette>.

The solver is entirely written in Python, but by mapping resource-intensive operations to large *NumPy* arrays, the *heavy lifting* is actually done in C on contiguous memory. Our code includes an implementation of KYBER, which uses symmetric primitives from the *PyCryptodome* library. Test routines compare the private key k_{priv} , the public key k_{pub} , the ciphertext c , and the shared secret k against those from the NIST reference implementation. To make all plots in this section reproducible, we include the methods that generated their data points, besides the solver itself.

6.1 Reduced Computation Time

The high computation time from previous solvers was already attributed to a single culprit, i.e., Eq. (10). We accelerate Eq. (10) by replacing the exact approach with an approximation. Considering that a large number of variables, i.e., $\psi - 1$, is being summed, the PMF of the sum can accurately be approximated by a normal distribution according to the *central limit theorem* (CLT). In later iterations, the binomial distributions evolved towards *one-point distributions*, and the approximation becomes less precise, but by then the algorithm is already honed in on the solution anyway. The resulting computation in Eq. (15) is light and straightforward. The summand $1/2$ compensates for the fact that a discrete distribution with step size 1 is approximated by a continuous distribution.

$$\mathbf{P}[i, j, k] \approx F_{\text{norm}} \left(\frac{\mathbf{A}[i, j] (k - \epsilon_1) + \left(\sum_{j' \in [0, \psi-1] \setminus \{j\}} \mathbf{A}[i, j'] \mathbf{E}[\mathbf{x}[j']] \right) + \mathbf{b}[i] + \frac{1}{2}}{\sqrt{\sum_{j' \in [0, \psi-1] \setminus \{j\}} \mathbf{A}[i, j']^2 \text{Var}[\mathbf{x}[j']]}} \right). \quad (15)$$

Instead of the reported 15 minutes, a single-threaded KYBER768 iteration with $\omega = 7000$ inequalities and $\psi = 1536$ unknowns now takes less than five seconds. These numbers are obtained from different computers, but as our number comes from a laptop with Python running in a *virtual machine*, we are unlikely to have a significant advantage. The need for parallelizing computations through threading is removed. In the next section on error tolerance, the benefit of the CLT-based acceleration increases, given that the required number of inequalities ω increases with the error rate.

6.2 Increased Error Tolerance

Our error-tolerant solver is represented in Algorithm 9. Whilst observed decapsulation successes are presumed to be 100% correct, observed decapsulation failures are only assumed to be correct up to a probability that is estimated in Line 4. Regarding the CBD in Line 3, we point out that the PMF of $e \triangleq e_1 - e_2$ where $e_1, e_2 \sim B(\epsilon_1, 1/2)$ can simply be evaluated as $f_{\text{bino}}(\epsilon_1 + e; 2\epsilon_1, 1/2)$, as proven in Appendix C.5. Not equally compact, Hermelink [HPP21b] loops over all pairs $(e_1, e_2) \in [0, \epsilon_1]^2$. As the probabilities $\mathbf{P}[i, j, k]$ might be small, the product in Line 13 is realized through a sum of logarithms to avoid underflow. Line 12 ensures that the logarithms do not receive inputs close to zero. The *stop criterion* in Line 6 is met if a maximum of 16 iterations is reached, or if a fitness value obtained from filling in \mathbf{x} in the inequalities does not improve anymore.

$$\Pr \left(\sum_{j=0}^{\psi-1} \mathbf{A}[i, j] \mathbf{x}[j] + \mathbf{b}[i] \geq 0 \right) \approx F_{\text{norm}} \left(\frac{\left(\sum_{j=0}^{\psi-1} \mathbf{A}[i, j] \mathbf{E}[\mathbf{x}[j]] \right) + \mathbf{b}[i] + \frac{1}{2}}{\sqrt{\sum_{j=0}^{\psi-1} \mathbf{A}[i, j]^2 \text{Var}[\mathbf{x}[j]]}} \right). \quad (16)$$

As noted in the IndoCrypt paper [HPP21a], correctly guessing $\psi/2$ out of ψ unknowns suffices for key-recovery, because the remaining half can be recovered via the public key k_{pub} . The authors implemented several confidence measures to select $\psi/2$ coefficients in every iteration, but we do not despite the reduction in the number of inequalities needed.

An alternative to our error-tolerant solver would be to leverage the asymmetric error rates of the IndoCrypt attack [HPP21a] and our roulette attacks by only retaining inequalities that correspond to an observed decapsulation success. However, discarding all other inequalities would be a waste of experimental data, and cannot as reliably be applied to the CHES attack [PP21a].

Algorithm 9 Solver

Input: Matrix \mathbf{A} with dimensions $\omega \times \psi$
Input: Vector \mathbf{b} of length ω
Input: Observations $\mathbf{r} \in \{0, 1\}^\omega$ where $\mathbf{r}[i] = 1$ if decapsulation fails and 0 otherwise
Output: Solution $\mathbf{x} \in [-\epsilon_1, \epsilon_1]^\psi$

- 1: **for** $j \in [0, \psi - 1]$ **do**
- 2: **for** $k \in [0, 2\epsilon_1]$ **do**
- 3: $f_{\mathbf{x}[j]}[k] \leftarrow f_{\text{bino}}(k; 2\epsilon_1, 1/2)$ ▷ Initialize PMFs
- 4: $\mathbf{p}_{\text{fail}} \leftarrow \mathbf{r} \cdot \min\left(\sum_{i=0}^{\omega-1} \mathbf{p}_{\text{fail}}[i] / \sum_{i=0}^{\omega-1} \mathbf{r}[i], 1\right)$ using Eq. (16) ▷ CLT
- 5: $\mathbf{x} \leftarrow (0 \ 0 \ \dots \ 0)$
- 6: **while** StopCriterionFails($\mathbf{A}, \mathbf{b}, \mathbf{r}, \mathbf{x}, \dots$) **do**
- 7: **for** $j \in [0, \psi - 1]$ **do**
- 8: **for** $k \in [0, 2\epsilon_1]$ **do**
- 9: **for** $i \in [0, \omega - 1]$ **do**
- 10: Compute $\mathbf{P}[i, j, k]$ in Eq. (15) ▷ CLT
- 11: $\mathbf{P}[i, j, k] \leftarrow \mathbf{P}[i, j, k] \mathbf{p}_{\text{fail}}[i] + (1 - \mathbf{P}[i, j, k])(1 - \mathbf{p}_{\text{fail}}[i])$
- 12: $\mathbf{P}[i, j, k] \leftarrow \max(\mathbf{P}[i, j, k], 10^{-5})$
- 13: $f'_{\mathbf{x}[j]}[k] = f_{\mathbf{x}[j]}[k] \prod_{i=0}^{\omega-1} \mathbf{P}[i, j, k]$ ▷ Sum of logarithms
- 14: $f'_{\mathbf{x}[j]} \leftarrow f'_{\mathbf{x}[j]} / \sum_{k=0}^{2\epsilon_1} f'_{\mathbf{x}[j]}[k]$ ▷ Normalization
- 15: $\mathbf{x}[j] \leftarrow -\epsilon_1 + \arg \max_{k \in [0, 2\epsilon_1]} f'_{\mathbf{x}[j]}[k]$ ▷ Largest mass
- 16: $f_{\mathbf{x}[j]} \leftarrow f'_{\mathbf{x}[j]}$

6.3 Experiments with Software-Simulated Faults

We perform three experiments where faults are simulated in software. Success is quantified by estimating the probability that any coefficient of the guessed solution \mathbf{x} is correct as a function of the provided number of inequalities, ω . This estimate is an average over (i) all ψ unknowns and (ii) 10 systems of inequalities that correspond to different key pairs ($k_{\text{pub}}, k_{\text{priv}}$). No runs are discarded, thereby demonstrating the stability of our solver.

In our first experiment, we revisit a filtering technique from Pessl and Prokop [PP21a] where inequalities are selected such that coefficient b is small in absolute value. This way, the probability of a decapsulation success (or failure) is approximately 50%. Hence, the information or Shannon entropy carried by the inequality is maximized, and fewer faults are needed for key recovery. Because the potential gains have not been quantified before, we do so in Fig. 4. For the unfiltered curve, the faulted ciphertext index $\iota \in [0, \eta - 1]$ is constant, and the result of a single encapsulation is unconditionally accepted. For the filtered curve, a single encapsulation is still performed, but the faulted index ι is variable and chosen such that $|b|$ is minimized. Remark that in an attack with actual hardware, a similar effect could be obtained by fixing ι and performing η encapsulations. Considering that the gains are significant, we filter inequalities by default.

In our second experiment, all three security levels of KYBER are compared in Fig. 5. The curves lie relatively close to one another, especially KYBER512 and KYBER768. This is at least partially attributable to the followings effects cancelling out: KYBER768 has more unknowns ($1536 > 1024$), whereas KYBER512 has more possible values per unknown ($7 > 5$). More rigorously, the Shannon entropy of the secret \mathbf{x} in Eq. (17) is roughly 2389, 3119, and 4159 bits for KYBER512, KYBER768, and KYBER1024 respectively, and provides a lower bound on the number of inequalities needed for a 100% success rate.

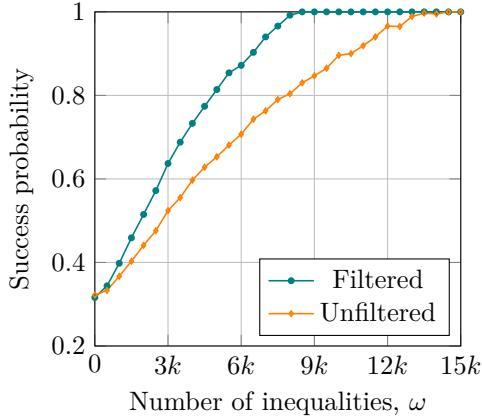


Figure 4: Solving filtered and unfiltered inequalities for KYBER512.

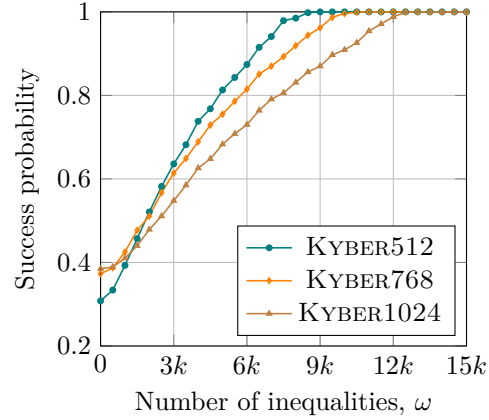


Figure 5: Solving filtered inequalities for all three security levels.

$$H[\mathbf{x}] = -\psi \sum_{k=0}^{2\epsilon_1} f_{\text{bino}}(k; 2\epsilon_1, 1/2) \log_2(f_{\text{bino}}(k; 2\epsilon_1, 1/2)). \quad (17)$$

In our third experiment, inequalities are corrupted. Given ω otherwise correct inequalities, decapsulation successes are turned into decapsulation failures with probability $p_{s2f} \in [0, 0.6]$, whereas decapsulation failures are untouched, which is in line with the working principles of the attack. Figure 6 shows that even with $p_{s2f} = 50\%$, the entire secret can still be recovered. The overall error rate is approximately half of p_{s2f} , resulting in an error tolerance of 25%. This is a considerable improvement upon the 1% reported by Pessl and Prokop [PP21a], and demands on the fault-injection setup are reduced accordingly.

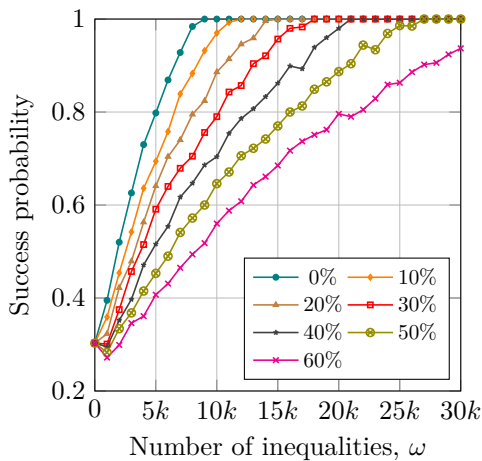


Figure 6: Solving corrupted inequalities for KYBER512.

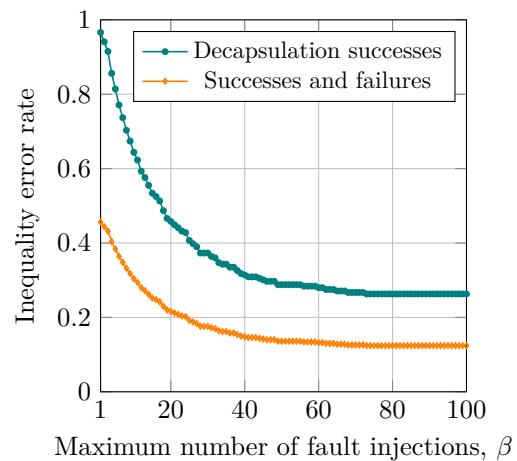


Figure 7: Inequality error rates obtained using a ChipWhisperer board.

7 ChipWhisperer Experiments

We experiment with actual fault-injection equipment and target a masked software implementation of KYBER running on an ARM Cortex-M4. Upon discarding (i) the *pqm4* implementation [KRSS] for being unprotected, (ii) the implementation of Bos et al. [BGR⁺21] for being closed-source, (iii) the implementation of Heinz et al. [HKL⁺22] for masking polynomial coefficients in \mathbb{Z}_ρ by sampling from $[0, 2^{12} - 1]$ instead of $[0, 3329 - 1]$ at the time of writing this paper, and (iv) the implementation of Bronchain and Cassiers [BC22] for storing the private key s and the symmetric key k in unmasked form at the time of writing this paper, we opted for the implementation of Coron et al. [CGMZ22]. Because the latter implementation is entirely written in plain C and thus unoptimized for the M4, it runs too slow for bulk experiments yet fast enough to show that our attack works. We build KYBER768 with first-order masking using *GNU Compiler Collection* (GCC) with $O3$ optimization.

We use a ChipWhisperer board from NewAE Technology Inc. [O’F] to generate and glitch a 24 MHz clock. Through a CW308 UFO Target Board, this clock is provided to the M4 that is contained in an STM32F405RGT6 chip from STMicroelectronics, and causes either instruction skips or instruction corruptions [TSW16]. The glitch is created by XORing a single short pulse with an otherwise proper clock signal, and is configured by three parameters: a global offset expressed as a number of clock cycles, a local offset with respect to the clock edge, and the width of the pulse. The latter two parameters jointly embody an *intensity* that must be carefully balanced for the given STM chip: if too low, no data is faulted, and if too high, our target crashes. The former parameter must be paired with a vulnerable spot of KYBER’s re-encryption and the given ciphertext index $\iota \in [0, 255]$ that is manipulated, which can be considered as a fourth parameter. Considering that we focused on the last layer of the INTT earlier-on, we mark this section of the source code with a trigger signal. Through a series of grid searches within the trigger window, four parameter values are selected. The selected ciphertext index $\iota = 130$. Remark that in a typical closed-source commercial product, a trigger cannot simply be added to the source code but may be derived from SCA or communications with chip peripherals such as external memory.

Upon selecting parameters, key recovery would be possible in a few hours up to a day for a well-optimized implementation of KYBER, but as we had to settle for an unoptimized target, it would take approximately five days. And ideally, multiple recoveries should be performed. Therefore, our attack is showcased through faster but fairly equivalent means: the ability to generate correct inequalities is measured. Based on 500 inequalities, Fig. 7 shows the probability of assigning the wrong sign to an inequality as a function of the maximum number of fault injections, β . Recall that only decapsulation successes can be misclassified and thus negatively contribute to the error rate. If, guided by Fig. 6, we tolerate misclassifying approximately 50% of the decapsulation successes, it should roughly hold that $\beta \geq 20$. To conclude: even with a cheap setup and an SCA-protected target, we can deliver a solvable system of inequalities.

8 Concluding Remarks

We overhauled a fault attack against KYBER proposed at IndoCrypt 2021 [HPP21a] such that it becomes easier to perform and harder to defend against. Firstly, popular masking techniques against SCA that originally favored the defender now favor the attacker. Secondly, more building blocks can be attacked, thereby increasing expenses for the defender. Thirdly, defending against a nearly perfect laser setup is no longer enough because cheaper methods such as voltage and clock glitching also suffice, even if they provide error-prone inequalities. Because off-the-shelf countermeasures such as executing the re-encryption

and ciphertext comparison twice merely decelerate our attack at a considerable cost, the design of more effective and affordable countermeasures is a first suggestion for follow-up work. In fact, Ravi et al. [RCB22] already responded with a custom countermeasure.

A second suggestion for follow-up work is the investigation of other PQC schemes. The authors of the IndoCrypt paper [HPP21a] already conjectured that a similar attack applies to SABER [BMD⁺20], which is another lattice-based KEM although not selected for standardization by NIST. Similarly, we conjecture that our roulette attacks can be mapped to SABER too. In the ideal case, a ciphertext coefficient $c_m \in \{0, 1\}^\tau$, where τ equals 3, 4, and 6 for LIGHTSABER, SABER, and FIRESABER respectively, is faulted such that c_m^* is uniformly distributed on $\{0, 1\}^\tau$. Furthermore, c_m^* is the result of rounding (pruning the least significant bits) and an addition, both of which are balanced functions as defined in Definition 2, i.e., the attack surface is large once again.

References

- [ABCG20] Erdem Alkim, Yusuf Alper Bilgin, Murat Cenk, and François Gérard. Cortex-M4 optimizations for {R,M}LWE schemes. *IACR Transactions on Cryptographic Hardware and Embedded Systems (TCHES)*, 2020(3):336–357, June 2020.
- [ABD⁺20] Roberto Avanzi, Joppe Bos, Léo Ducas, Eike Kiltz, Tancrede Lepoint, Vadim Lyubashevsky, John M. Schanck, Peter Schwabe, Gregor Seiler, and Damien Stehlé. CRYSTALS-Kyber: Algorithm specifications and supporting documentation. Technical report, National Institute of Standards and Technology (NIST), October 2020. <https://csrc.nist.gov/projects/post-quantum-cryptography/round-3-submissions>.
- [ASMM18] Ravi Anand, Akhilesh Siddhanti, Subhamoy Maitra, and Sourav Mukhopadhyay. Differential fault attack on SIMON with very few faults. In Debrup Chakraborty and Tetsu Iwata, editors, *19th International Conference on Cryptology in India (INDOCRYPT 2018)*, volume 11356 of *Lecture Notes in Computer Science*, pages 107–119. Springer, December 2018.
- [BC22] Olivier Bronchain and Gaëtan Cassiers. Bitslicing arithmetic/boolean masking conversions for fun and profit with application to lattice-based KEMs. *Cryptology ePrint Archive*, Report 2022/158, 2022. <https://ia.cr/2022/158>, https://github.com/uclcrypto/pqm4_masked, commit ed6ab2f55f407332b36bee5c29c828778c1266c0.
- [BGR⁺21] Joppe W. Bos, Marc Gourjon, Joost Renes, Tobias Schneider, and Christine van Vredendaal. Masking Kyber: First- and higher-order implementations. *IACR Transactions on Cryptographic Hardware and Embedded Systems (TCHES)*, 2021(4):173–214, August 2021.
- [BH08] Arnaud Boscher and Helena Handschuh. Masking does not protect against differential fault attacks. In Luca Breveglieri, Shay Gueron, Israel Koren, David Naccache, and Jean-Pierre Seifert, editors, *5th Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC 2008)*, pages 35–40. IEEE, August 2008.
- [BMD⁺20] Andrea Basso, Jose Maria Bermudo Mera, Jan-Pieter D’Anvers, Angshuman Karmakar, Sujoy Sinha Roy, Michiel Van Beirendonck, and Frederik Vercauteren. SABER: Mod-LWR based KEM (round 3 submission). Technical report, National Institute of Standards and Technology (NIST), October

2020. <https://csrc.nist.gov/projects/post-quantum-cryptography/round-3-submissions>.
- [BMR21] Luk Bettale, Simon Montoya, and Guénaél Renault. Safe-error analysis of post-quantum cryptography mechanisms. In *18th Workshop on Fault Detection and Tolerance in Cryptography (FDTC 2021)*, pages 39–44. IEEE, September 2021.
- [CGMZ22] Jean-Sébastien Coron, François Gérard, Simon Montoya, and Rina Zeitoun. High-order polynomial comparison and masking lattice-based encryption. GitHub repository, <https://github.com/fragerar/HOTableConv>, commit 5d493b68b0b7ac3d41d0908bb99a0705d13ce20c, January 2022.
- [Cla07] Christophe Clavier. Secret external encodings do not prevent transient fault analysis. In Pascal Paillier and Ingrid Verbauwhede, editors, *9th Workshop on Cryptographic Hardware and Embedded Systems (CHES 2007)*, volume 4727 of *Lecture Notes in Computer Science*, pages 181–194. Springer, September 2007.
- [DEG⁺18] Christoph Dobraunig, Maria Eichlseder, Hannes Groß, Stefan Mangard, Florian Mendel, and Robert Primas. Statistical ineffective fault attacks on masked AES with fault countermeasures. In Thomas Peyrin and Steven D. Galbraith, editors, *Advances in Cryptology - ASIACRYPT 2018*, volume 11273 of *Lecture Notes in Computer Science*, pages 315–342. Springer, December 2018.
- [DEK⁺18] Christoph Dobraunig, Maria Eichlseder, Thomas Korak, Stefan Mangard, Florian Mendel, and Robert Primas. SIFA: exploiting ineffective fault inductions on symmetric cryptography. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2018(3):547–572, August 2018.
- [Del20] Jeroen Delvaux. Threshold implementations are not provably secure against fault sensitivity analysis. Cryptology ePrint Archive, Report 2020/400, 2020. <https://ia.cr/2020/400>.
- [DMB19] Lauren De Meyer and Begül Bilgin. Classification of balanced quadratic functions. *IACR Transactions on Symmetric Cryptology*, 2019(2):169–192, 2019.
- [DTVV19] Jan-Pieter D’Anvers, Marcel Tiepelt, Frederik Vercauteren, and Ingrid Verbauwhede. Timing attacks on error correcting codes in post-quantum schemes. In Begül Bilgin, Svetla Petkova-Nikova, and Vincent Rijmen, editors, *Proceedings of ACM Workshop on Theory of Implementation Security Workshop (TIS@CCS 2019)*, pages 2–9. ACM, November 2019.
- [HKL⁺22] Daniel Heinz, Matthias J. Kannwischer, Georg Land, Thomas Pöppelmann, Peter Schwabe, and Daan Sprenkels. First-order masked Kyber on ARM Cortex-M4. GitHub repository, <https://github.com/masked-kyber-m4/mkm4>, commit b480a071459881531b6bf8b42410b99f94d637fe, January 2022.
- [HPP21a] 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*, volume 13143 of *Lecture Notes in Computer Science*, pages 311–334. Springer, December 2021.

- [HPP21b] Julius Hermelink, Peter Pessl, and Thomas Pöppelmann. Fault-enabled chosen-ciphertext attacks on Kyber. GitHub repository, https://github.com/juliusjh/fault_enabled_cca, commit f70b246dd45841251f3f63a55c084fd191380efb, December 2021.
- [KRSS] Matthias J. Kannwischer, Joost Rijneveld, Peter Schwabe, and Ko Stoffelen. PQM4: Post-quantum crypto library for the ARM Cortex-M4. <https://github.com/mupq/pqm4>, commit 8970d37a8c3055d9579007e954449d926c3583b4 (January 20, 2020).
- [KWMK02] Ramesh Karri, Kaijie Wu, Piyush Mishra, and Yongkook Kim. Concurrent error detection schemes for fault-based side-channel cryptanalysis of symmetric block ciphers. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 21(12):1509–1517, December 2002.
- [LOS12] Yang Li, Kazuo Ohta, and Kazuo Sakiyama. New fault-based side-channel attack using fault sensitivity. *IEEE Transactions on Information Forensics and Security*, 7(1):88–97, February 2012.
- [MMP⁺11] Amir Moradi, Oliver Mischke, Christof Paar, Yang Li, Kazuo Ohta, and Kazuo Sakiyama. On the power of fault sensitivity analysis and collision side-channel attacks in a combined setting. In Bart Preneel and Tsuyoshi Takagi, editors, *13th Workshop on Cryptographic Hardware and Embedded Systems (CHES 2011)*, volume 6917 of *Lecture Notes in Computer Science*, pages 292–311. Springer, October 2011.
- [O’F] Colin O’Flynn. NewAE Technology Inc. <https://www.newae.com>. Accessed: June 14, 2022.
- [OSPG18] Tobias Oder, Tobias Schneider, Thomas Pöppelmann, and Tim Güneysu. Practical CCA2-secure and masked ring-LWE implementation. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2018(1):142–174, February 2018.
- [PP21a] Peter Pessl and Lukás Prokop. Fault attacks on CCA-secure lattice KEMs. *IACR Transactions on Cryptographic Hardware and Embedded Systems (TCHES)*, 2021(2):37–60, February 2021.
- [PP21b] Peter Pessl and Lukás Prokop. Fault attacks on CCA-secure lattice KEMs. GitHub repository, <https://github.com/latticekemfaults/latticekemfaults/>, commit 85f78d87218e6fd4f5366c416b5db4a8ecfa65fb, April 2021.
- [RCB22] Prasanna Ravi, Anupam Chattopadhyay, and Anubhab Baksi. Side-channel and fault-injection attacks over lattice-based post-quantum schemes (Kyber, Dilithium): Survey and new results. *Cryptology ePrint Archive*, Paper 2022/737, June 2022. <https://eprint.iacr.org/2022/737>.
- [RPBC20] Prasanna Ravi, Romain Poussier, Shivam Bhasin, and Anupam Chattopadhyay. On configurable SCA countermeasures against single trace attacks for the NTT. In Lejla Batina, Stjepan Picek, and Mainack Mondal, editors, *10th Conference on Security, Privacy, and Applied Cryptography Engineering (SPACE 2020)*, volume 12586 of *Lecture Notes in Computer Science*, pages 123–146. Springer, December 2020.

- [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, *10th Workshop on Constructive Side-Channel Analysis and Secure Design (COSADE 2019)*, volume 11421 of *Lecture Notes in Computer Science*, pages 232–250. Springer, April 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 Transactions on Cryptographic Hardware and Embedded Systems (TCHES 2020)*, 2020(3):307–335, June 2020.
- [Saa18] Markku-Juhani O. Saarinen. Arithmetic coding and blinding countermeasures for lattice signatures. *Journal of Cryptographic Engineering*, 8(1):71–84, April 2018.
- [SH07] Jörn-Marc Schmidt and Michael Hutter. Optical and EM fault-attacks on CRT-based RSA. In *15th Austrian Workshop on Microelectronics (Austrochip 2007)*, pages 61–67. Technischen Universität Graz, October 2007.
- [TSW16] Niek Timmers, Albert Spruyt, and Marc Witteman. Controlling PC on ARM using fault injection. In *13th Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*, pages 25–35. IEEE, August 2016.
- [UXT⁺21] Rei Ueno, Keita Xagawa, Yutaro Tanaka, Akira Ito, Junko Takahashi, and Naofumi Homma. Curse of re-encryption: A generic power/EM analysis on post-quantum KEMs. *IACR Transactions on Cryptographic Hardware and Embedded Systems (TCHES 2022)*, 2022(1):296–322, November 2021.
- [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, *Fifth Workshop on Cryptography and Security in Computing Systems (CS2 2018)*, pages 7–12. ACM, January 2018.
- [XIU⁺21] Keita Xagawa, Akira Ito, Rei Ueno, Junko Takahashi, and Naofumi Homma. Fault-injection attacks against NIST’s post-quantum cryptography round 3 KEM candidates. In *Advances in Cryptology – ASIACRYPT 2021*, volume 13091 of *Lecture Notes in Computer Science*, pages 33–61. Springer, December 2021.
- [YJ00] Sung-Ming Yen and Marc Joye. Checking before output may not be enough against fault-based cryptanalysis. *IEEE Transactions on Computers*, 49(9):967–970, September 2000.

A Omitting HD Constraint in Kyber1024

Table 5: Properties of the compressed ciphertext coefficients $v \in [0, 2^\delta - 1]$ where $\delta = 5$.

		Original			Manipulated			
Bin	Size	First	Last	Center	Bin	Fault	HD	Δm^*
0	105	3277	52	0	8			
1		53	156	104	9			
2		157	260	208	10			
3		261	364	312	11	01000	1	$\Delta m + 832$
4		365	468	416	12			
5		469	572	520	13			
6		573	676	624	14			
7		677	780	728	15			
8		781	884	832	16			
9		885	988	936	17			
10		989	1092	1040	18			
11		1093	1196	1144	19	11000	2	$\Delta m + 833$
12		1197	1300	1248	20			
13		1301	1404	1352	21			
14		1405	1508	1456	22			
15	104	1509	1612	1560	23			
16		1613	1716	1665	24			
17		1717	1820	1769	25			
18		1821	1924	1873	26			
19		1925	2028	1977	27	01000	1	$\Delta m + 832$
20		2029	2132	2081	28			
21		2133	2236	2185	29			
22		2237	2340	2289	30			
23		2341	2444	2393	31			
24		2445	2548	2497	0			
25		2549	2652	2601	1			
26		2653	2756	2705	2			
27		2757	2860	2809	3	11000	2	$\Delta m + 833$
28		2861	2964	2913	4			
29		2965	3068	3017	5			
30		3069	3172	3121	6			
31		3173	3276	3225	7			

B Roulette Attack on Decryption Module

Section 5.2 specified a first roulette attack on KYBER’s decapsulation, in which the re-encryption is faulted in order to recover the private key s . This appendix specifies a second roulette attack on the decapsulation, but now the decryption is faulted in order to recover the message m and the corresponding session key k . This second attack is much more ‘academic’ because (i) the distribution of the faulted value must be known, and (ii) millions of perfectly injected faults are required. Nevertheless, there is no harm in reporting an exploit on building blocks that have not previously been faulted, even if it only serves as a reminder that not only obvious targets such as the polynomial comparison

should be protected.

The generic variable $z \in \mathcal{Z}$ in Fig. 2 is instantiated with an uncompressed message coefficient $m \in [0, \rho-1]$. Although practically any distribution of its faulted counterpart m^* enables the attack, at least if the distribution is known to the attacker, we again idealize the case where m^* is uniformly distributed on $[0, \rho-1]$. Leveraging fault propagation, the attack surface consists of $\text{Decompress}(v; \rho, \delta_v)$, a butterfly in the final layer of the INTT, and a modular subtraction. Recall that the modular subtraction is balanced according to Lemma 3, i.e., a uniformly distributed fault in the butterfly or decompression output results in a uniformly distributed m^* . Given that primes ρ are odd, the final decryption step, i.e., $m^* \leftarrow \text{Compress}(m^*; \rho, 1)$ as defined in Eq. (4), is inherently biased. As illustrated in Fig. 8 for $\rho = 7$, the compression function maps $\lfloor \rho/2 \rfloor = 3$ coefficients in $[0, \rho-1]$ to $m^* = 0$, whereas $\lceil \rho/2 \rceil = 4$ coefficients map to $m^* = 1$.

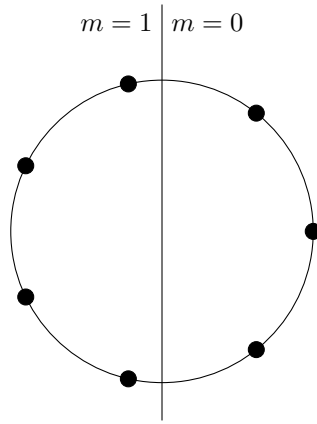


Figure 8: Message coefficients m before and after compression according to Eq. (4) where prime $\rho = 7$.

For the actual prime $\rho = 3329$ used in KYBER, the right and left semicircles contain $\lfloor \rho/2 \rfloor = 1664$ and $\lceil \rho/2 \rceil = 1665$ field elements respectively. Hence, the probability of a failed decapsulation is $1665/3329 \approx 50.015\%$ if the original message bit $m = 0$ and $1664/3329 \approx 49.985\%$ otherwise. At least in theory, a measurement of this failure rate suffices to recover m . For $\beta = 18201189$ perfectly faulted decapsulations, the recovery succeeds with 90% certainty, as can be derived from the *cumulative distribution function* (CDF) of a binomial distribution: $F_{\text{bino}}(\lfloor \beta/2 \rfloor; \beta, 1664/3329) \geq 90\%$ where n is odd. Apart from the staggering number of faults, the attack is hampered in practice because fault injections are unlikely to be perfect, and the probability that no fault is injected is typically unknown.

C Proofs

C.1 Lemma 2

The case $G : \mathcal{X} \rightarrow \mathcal{Y}$ of Lemma 2 is proven in Eq. (18); the case $G : \mathcal{X}_1 \times \mathcal{X}_2 \rightarrow \mathcal{Y}$ is proven in Eq. (19).

$$\Pr(y) = \sum_{\substack{x \in \mathcal{X} \\ G(x)=y}} \Pr(x) = \frac{|\mathcal{X}|}{|\mathcal{Y}|} \cdot \frac{1}{|\mathcal{X}|} = \frac{1}{|\mathcal{Y}|}. \quad (18)$$

$$\begin{aligned}
 \Pr(y) &= \sum_{\substack{(x_1, x_2) \in \mathcal{X}_1 \times \mathcal{X}_2 \\ \text{s.t. } G(x_1, x_2) = y}} \Pr(x_1 \wedge x_2) = \sum_{x_2 \in \mathcal{X}_2} \Pr(x_2) \sum_{\substack{x_1 \in \mathcal{X}_1 \text{ s.t.} \\ G(x_1, x_2) = y}} \Pr(x_1) \\
 &= \frac{|\mathcal{X}_1|}{|\mathcal{Y}|} \cdot \frac{1}{|\mathcal{X}_1|} \cdot \sum_{x_2 \in \mathcal{X}_2} \Pr(x_2) = \frac{1}{|\mathcal{Y}|}.
 \end{aligned} \tag{19}$$

C.2 Lemma 3

Balancedness with respect to input $x_1 \in \mathcal{R}$ in Lemma 3 is proven in Eq. (20) and follows from the property that each element in a ring has an *additive inverse*. Balancedness with respect to input $x_2 \in \mathcal{R}$ is proven in an identical manner.

$$\forall (x_2, y) \in \mathcal{R}^2, |\{x_1 \in \mathcal{R} \mid x_1 + x_2 = y\}| = |\{y - x_2\}| = 1. \tag{20}$$

C.3 Lemma 4

Balancedness with respect to input $x_1 \in \mathcal{F}$ in Lemma 4 is proven in Eq. (21) and follows from the property that each element $x_2 \neq 0$ in a field has a *multiplicative inverse*.

$$\forall (x_2, y) \in \mathcal{F}^2, |\{x_1 \in \mathcal{F} \mid x_1 \cdot x_2 = y\}| = |\{y \cdot x_2^{-1}\}| = 1. \tag{21}$$

C.4 Instruction Skips in Double Butterflies on ARM Cortex M4

To prove uniformity, we start from the observation that for each out of λ shares, the input to last INTT layer is uniformly distributed on $\mathbb{Z}_\rho^\eta = \mathbb{Z}_\rho^{256}$, which implies that all 256 finite-field elements are independent of one another. This follows from Lemma 1 and the fact that every INTT layer is a permutation on \mathbb{Z}_ρ^η .

The proofs for instructions 2 to 4 in Table 4 are particularly straightforward and given in Eqs. (22) to (24) respectively. Note that the faulty output shares are low in magnitude even before being reduced by the Montgomery macro and cannot violate the margin for lazy reductions in any building block following the double butterfly. Also, note that the multiplications with τ , $(\tau + 1)$, or $(1 - \tau)$ preserve uniformity according to Lemma 4.

$$\left\{ \begin{array}{l} (c_1^{(1)}, c_2^{(1)})^* = (a_1^{(1)}, a_2^{(1)}), \\ (c_1^{(2)}, c_2^{(2)}) = (a_1^{(2)} + b_1^{(2)}, a_2^{(2)} + b_2^{(2)}), \\ \vdots \\ (c_1^{(\lambda)}, c_2^{(\lambda)}) = (a_1^{(\lambda)} + b_1^{(\lambda)}, a_2^{(\lambda)} + b_2^{(\lambda)}), \end{array} \right. \implies (c_1, c_2)^* = (c_1, c_2) - (b_1^{(1)}, b_2^{(1)}) \sim U(\mathbb{Z}_\rho^2). \tag{22}$$

$$\left\{ \begin{array}{l} (d_1^{(1)})^* = b_1^{(1)}, \\ d_1^{(2)} = (a_1^{(2)} - b_1^{(2)})\tau, \\ \vdots \\ d_1^{(\lambda)} = (a_1^{(\lambda)} - b_1^{(\lambda)})\tau, \end{array} \right. \implies d_1^* = d_1 + b_1^{(1)}(\tau + 1) - a_1^{(1)}\tau \sim U(\mathbb{Z}_\rho). \tag{23}$$

$$\begin{cases} (d_2^{(1)})^* = a_2^{(1)} - b_2^{(1)}, \\ d_2^{(2)} = (a_2^{(2)} - b_2^{(2)})\tau, \\ \vdots \\ d_2^{(\lambda)} = (a_2^{(\lambda)} - b_2^{(\lambda)})\tau \end{cases} \implies d_2^* = d_2 + (a_2^{(1)} - b_2^{(1)})(1 - \tau) \sim U(\mathbb{Z}_\rho). \quad (24)$$

For instruction 1 in Table 4, the faulted output coefficients $(d_1, d_2)^*$ are determined by an uninitialized temporary variable t_1 , as formalized in Eq. (25). Following the INTT implementation of the *pqm4* library, each layer is completed before starting the next one, and for the most part, t_1 has last been set in another double butterfly in the last layer. Hence, t_1 is independent of the current double-butterfly inputs. As for instructions 2 to 4, the faulted output shares are reduced by the Montgomery macro.

$$\begin{cases} (d_1^{(1)}, d_2^{(1)})^* = (t_1[15:0], t_1[31:16])\tau, \\ (d_1^{(2)}, d_2^{(2)}) = (a_1^{(2)} - b_1^{(2)}, a_2^{(2)} - b_2^{(2)})\tau, \\ \vdots \\ (d_1^{(\lambda)}, d_2^{(\lambda)}) = (a_1^{(\lambda)} - b_1^{(\lambda)}, a_2^{(\lambda)} - b_2^{(\lambda)})\tau, \end{cases} \quad \begin{array}{l} \text{where } t_1 \text{ and} \\ (a_1^{(1)}, b_1^{(1)}, a_2^{(1)}, b_2^{(1)}) \\ \text{are independent,} \end{array} \quad (25a)$$

$$\implies (d_1, d_2)^* = (d_1, d_2) + (t_1[15:0] - a_1^{(1)} + b_1^{(1)}, t_1[31:16] - a_2^{(1)} + b_2^{(1)})\tau \sim U(\mathbb{Z}_\rho^2) \quad (25b)$$

For instruction 7 in Table 4, the faulty output coefficient d_1^* is uniformly distributed on \mathbb{Z}_ρ in theory, but not necessarily in practice. The output of the function \mathbf{M} is not properly reduced, and the margin for lazy reduction may be violated in building blocks following the double butterfly. Such violations may still produce the desired result in Eq. (12), but are hard to analyze from a mathematical perspective and not further addressed here.

$$\begin{cases} (d_1^{(1)})^* = \mathbf{M}((a_2^{(1)} - b_2^{(1)})\tau), \\ d_1^{(2)} = (a_1^{(2)} - b_1^{(2)})\tau, \\ \vdots \\ d_1^{(\lambda)} = (a_1^{(\lambda)} - b_1^{(\lambda)})\tau \end{cases} \implies \begin{array}{l} d_1^* = d_1 + \mathbf{M}((a_2^{(1)} - b_2^{(1)})\tau) \\ -(a_1^{(1)} - b_1^{(1)})\tau \sim U(\mathbb{Z}_\rho). \end{array} \quad (26)$$

C.5 PMF of CBD

Let X be a random variable with a CBD, i.e., $X \triangleq X_1 - X_2$ where $X_1, X_2 \sim B(\epsilon, 1/2)$. The PMF of X is derived in Eq. (27). Vandermonde's identity is used in Eq. (27c) to dispose of the summation operator.

$$\Pr(X = i) = \Pr(X = |i|) = \sum_{j=0}^{\epsilon-|i|} \Pr(X_2 = j) \Pr(X_1 = |i| + j) \quad (27a)$$

$$= \sum_{j=0}^{\epsilon-|i|} \mathbf{f}_{\text{bino}}(j; \epsilon, 1/2) \mathbf{f}_{\text{bino}}(|i| + j; \epsilon, 1/2) = \sum_{j=0}^{\epsilon-|i|} \binom{\epsilon}{j} \left(\frac{1}{2}\right)^\epsilon \binom{\epsilon}{|i| + j} \left(\frac{1}{2}\right)^\epsilon \quad (27b)$$

$$= \left(\frac{1}{2}\right)^{2\epsilon} \sum_{j=0}^{\epsilon-|i|} \binom{\epsilon}{j} \binom{\epsilon}{\epsilon-|i|-j} = \left(\frac{1}{2}\right)^{2\epsilon} \binom{2\epsilon}{\epsilon-|i|} = \mathbf{f}_{\text{bino}}(\epsilon-|i|; 2\epsilon, 1/2). \quad (27c)$$