Roulette: Breaking Kyber with Diverse Fault-Injection Setups

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* (NTT). 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, the IndoCrypt attack is like most other fault attacks either hindered or unaffected by countermeasures against passive side-channel attacks, i.e., masking and blinding of sensitive variables, whereas our attack is an exception to this rule. Given that we randomly fault prime-field elements until a desired set of values is hit, randomization-based countermeasures kindly help us with injecting randomness. If 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, at the time of writing this paper, a round-3 finalist in the on-going *post-quantum cryptography* (PQC) standardization process run by the United States' *National Institute of Standards and Technology* (NIST). Although the selection process initially focused on the mathematical strength and the implementation efficiency of the proposals, the resistance to *side-channel analysis* (SCA) and fault attacks became a major topic towards the end—round 3 in particular. 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.

The authors suggest using a laser owing to its small *spot size* and thus spatial precision. The time, budget, and expertise needed to decapsulate a chip [SH07] and calibrate 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, Hermelink et al. [HPP21a] 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* (NTT), 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 key recovery: roughly speaking, 1000s become 10000s or 100000s. Even so, for a well-optimized Kyber implementation that is clocked at MHz rates, the latter range typically equates to a day or less 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.

- Considering that Kyber has known weaknesses against *side-channel analysis* (SCA) [UXT+22], such as power-consumption analysis, countermeasures should be in place. We lay out a peculiar case where masking and certain forms of blinding facilitate a fault attack. Under normal circumstances, which includes the IndoCrypt paper [HPP21a], SCA countermeasures either decrease the vulnerability to fault attacks or result in a *status quo*. Because we fault otherwise input-defined prime-field elements such that they cover a wide range of values, ideally but not necessarily uniformly distributed, countermeasures that randomize intermediate variables 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 inverse NTT 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 the 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, e.g., F. With $\lceil \cdot \rfloor$, we denote rounding to the nearest integer where ties (fraction 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 $\mathbb{R}_{(\rho,\eta)} = \mathbb{Z}_\rho[x]/(x^\eta + 1)$, 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., $\zeta^{256} \mod \rho = 1$ 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 Equation (1), where operator $\circ$ comprises $\eta/2 = 128$ products of linear polynomials.

Algorithm 1 Kyber.PKE.KeyGen

Output: Public key p
Output: Private key $\hat{\mathbf{s}}$
1: $d \leftarrow \{0,1\}^{256}$
2: $(q, b) \leftarrow \mathsf{G}(d)$
3: **for** $i \in [0, \kappa - 1]$ **do**
4: **for** $j \in [0, \kappa - 1]$ **do**
5: $\hat{\mathbf{A}}[i, j] \leftarrow \mathsf{Parse}(\mathsf{XOF}(q; j, i))$
6: $\mathbf{s}[i] \leftarrow \mathsf{CBD}(\mathsf{PRF}(b; i); \epsilon_1)$
7: $\hat{\mathbf{s}}[i] \leftarrow \mathsf{NTT}(\mathbf{s}[i])$
8: $\mathbf{e}[i] \leftarrow \mathsf{CBD}(\mathsf{PRF}(b; \kappa + i); \epsilon_1)$
9: $\hat{\mathbf{e}}[i] \leftarrow \mathsf{NTT}(\mathbf{e}[i])$
10: $\hat{\mathbf{t}} \leftarrow \hat{\mathbf{A}} \circ \hat{\mathbf{s}} + \hat{\mathbf{e}}$
11: $p \leftarrow \hat{\mathbf{t}} \| q$

Algorithm 2 Kyber.PKE.Encrypt

Input: Public key $p \triangleq \hat{\mathbf{t}} \| q$
Input: Message m
Input: Coins r
Output: Ciphertext $c \triangleq (\mathbf{u}, v)$
1: **for** $i \in [0, \kappa - 1]$ **do**
2: **for** $j \in [0, \kappa - 1]$ **do**
3: $\hat{\mathbf{A}}^\intercal[i, j] \leftarrow \mathsf{Parse}(\mathsf{XOF}(q; i, j))$
4: $\mathbf{r}[i] \leftarrow \mathsf{CBD}(\mathsf{PRF}(r; i); \epsilon_1)$
5: $\hat{\mathbf{r}}[i] \leftarrow \mathsf{NTT}(\mathbf{r}[i])$
6: $\mathbf{e}_1 \leftarrow \mathsf{CBD}(\mathsf{PRF}(r; \kappa + i); \epsilon_2)$
7: $e_2 \leftarrow \mathsf{CBD}(\mathsf{PRF}(r; 2\,\kappa); \epsilon_2)$
8: $\hat{\mathbf{u}} \leftarrow \hat{\mathbf{A}}^\intercal \circ \hat{\mathbf{r}}$
9: **for** $i \in [0, \kappa - 1]$ **do**
10: $\mathbf{u}[i] \leftarrow \mathsf{INTT}(\hat{\mathbf{u}}[i]) + \mathbf{e}_1$
11: $\mathbf{u}[i] \leftarrow \mathsf{Compress}(\mathbf{u}[i]; \rho, \delta_u)$
12: $m \leftarrow \mathsf{Decompress}(m; \rho, 1)$
13: $v \leftarrow \mathsf{INTT}(\hat{\mathbf{t}}^\intercal \circ \hat{\mathbf{r}}) + e_2 + m$
14: $v \leftarrow \mathsf{Compress}(v; \rho, \delta_v)$

Algorithm 3 Kyber.PKE.Decrypt

Input: Private key $\hat{\mathbf{s}}$
Input: Ciphertext $c \triangleq (\mathbf{u}, v)$
Output: Message m
1: **for** $i \in [0, \kappa - 1]$ **do**
2: $\mathbf{u}[i] \leftarrow \mathsf{Decompress}(\mathbf{u}[i]; \rho, \delta_u)$
3: $\hat{\mathbf{u}}[i] \leftarrow \mathsf{NTT}(\mathbf{u}[i])$
4: $v \leftarrow \mathsf{Decompress}(v; \rho, \delta_v)$
5: $m \leftarrow v - \mathsf{INTT}(\hat{\mathbf{s}}^\intercal \circ \hat{\mathbf{u}})$
6: $m \leftarrow \mathsf{Compress}(m; \rho, 1)$

Algorithm 4 Kyber.KEM.KeyGen

Output: Public key p
Output: Private key s
1: $z \leftarrow \{0,1\}^{256}$
2: $(p, \hat{\mathbf{s}}) \leftarrow \mathsf{Kyber.PKE.KeyGen}()$
3: $h \leftarrow \mathsf{H}(p)$
4: $s \leftarrow \hat{\mathbf{s}} \| p \| h \| z$

Algorithm 5 Kyber.KEM.Encapsulate

Input: Public key p
Output: Ciphertext c
Output: Symmetric key k
1: $m \leftarrow \{0,1\}^{256}$
2: $m \leftarrow \mathsf{H}(m)$
3: $(k, r) \leftarrow \mathsf{G}(m \| \mathsf{H}(p))$
4: $c \leftarrow \mathsf{Kyber.PKE.Encrypt}(p, m, r)$
5: $k \leftarrow \mathsf{KDF}(k \| \mathsf{H}(c))$

Algorithm 6 Kyber.KEM.Decapsulate

Input: Ciphertext c
Input: Private key $s' \triangleq \hat{\mathbf{s}} \| p \| h \| z$
Output: Symmetric key k
1: $m \leftarrow \mathsf{Kyber.PKE.Decrypt}(\hat{\mathbf{s}}, c)$
2: $(k, r) \leftarrow \mathsf{G}(m \| h)$
3: $c' \leftarrow \mathsf{Kyber.PKE.Encrypt}(p, m, r)$
4: **if** $c = c'$ **then**
5: $k \leftarrow \mathsf{KDF}(k \| \mathsf{H}(c))$
6: **else**
7: $k \leftarrow \mathsf{KDF}(z \| \mathsf{H}(c))$

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

$$a[x] \cdot b[x] \mod (x^\eta + 1) = \mathsf{INTT}(\mathsf{NTT}(a) \circ \mathsf{NTT}(b)). \tag{1}$$

The NTT and the *inverse NTT* (INTT) both consist of $\log_2(\eta) - 1 = 7$ layers that each contains $\eta/2 = 128$ butterfly operations $\mathsf{Butterfly} : \mathbb{Z}_\rho^2 \to \mathbb{Z}_\rho^2$. 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 $\mathbb{Z}_\rho$.

$$\mathsf{GSButterfly}(a, b; \tau) \triangleq \big(a + b, (a - b)\,\tau\big) \mod \rho. \tag{2}$$

The lossy compression is defined in Eqs. (3) and (4). The compression of message coefficients $m \in \mathbb{Z}_\rho$ in Line 6 of Algorithm 3 uses $\delta = 1$, and Eq. (3) boils down to Eq. (5).

$$\mathsf{Compress}(x; \rho, \delta) = \lceil 2^\delta x/\rho \rfloor \mod 2^\delta. \tag{3}$$

$$\mathsf{Decompress}(x; \rho, \delta) = \lceil \rho\, x/2^\delta \rfloor. \tag{4}$$

$$\mathsf{Compress}(x; \rho, 1) = \begin{cases} 1 & \text{if } q/4 < x < 3q/4, \\ 0 & \text{otherwise.} \end{cases} \tag{5}$$

The decryption faces an accumulated error on the message $m \in \mathbb{R}_{(\rho,\eta)}$ as given in Eq. (6), where summands $\Delta\mathbf{u}$ and Δv denote contributions from the lossy ciphertext compression. If it holds for each coefficient $i \in [0, \eta - 1]$ that $-\rho/2 < \Delta m_i < \rho/2$, then the decryption outputs the correct message $m \in \{0,1\}^\eta$ after its final step $\mathsf{Compress}(\cdot; \rho, 1)$.

$$\Delta m = \mathbf{e}^\intercal \mathbf{r} - \mathbf{s}^\intercal(\mathbf{e}_1 + \Delta\mathbf{u}) + e_2 + \Delta v \mod^{\pm} \rho \tag{6}$$

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 G and H 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. 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. Algorithm 7 shows the latest version from the *pqm4* library, which only differs from the academic paper in how temporary variables are used. 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, commit on 20 Jan 2020]

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 $-\rho^{-1} = 3327$
Output: Reduced $t[31:16]$ where $-\rho < t[31:16] < \rho$

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

Algorithm 8 DoubleGSButterfly [KRSS, commit on 20 Jan 2020]

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

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 chosen-ciphertext attacks proposed by D'Anvers et al. [DTVV19], Ravi et al. [RRCB20], and Ueno et al. [UXT+22] subverted this intuition. An attacker can construct ciphertexts c such that the correctness of a single decrypted message bit $m \in \{0,1\}$ depends on $\hat{s}$. To avoid the realization of a message-checking oracle through SCA, algorithms that process m should be protected. This includes hash function G, the encryption, and the ciphertext comparison. 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 $\left(x^{(2)}, x^{(3)}, \cdots, x^{(\lambda)}\right)$ uniformly at random from $\mathcal{X}^{\lambda-1}$, followed by a computation $x^{(1)} = x - x^{(2)} - x^{(3)} - \cdots - x^{(\lambda)}$.

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

We distinguish between Boolean masking, where $\mathcal{X} = \{0,1\}^{\sigma}$ and additions are defined by XORing, and 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* (B2A) and *arithmetic-to-Boolean* (A2B) conversions are commonplace.

A function $\mathsf{F} : \mathcal{X} \rightarrow \mathcal{Y}$ must also be split such that shares of $x \in \mathcal{X}$ satisfying Definition 1 are mapped to shares of $y = \mathsf{F}(x)$ that again satisfy Definition 1. If F is linear, F is trivially split by defining $\forall i \in [1, \lambda] : \mathsf{F}^{(i)}(x^{(i)}) \triangleq \mathsf{F}(x^{(i)})$, considering that $\mathsf{F}^{(1)}(x^{(1)}) + \mathsf{F}^{(2)}(x^{(2)}) + \cdots + \mathsf{F}^{(\lambda)}(x^{(\lambda)}) = \mathsf{F}(x^{(1)} + x^{(2)} + \cdots + x^{(\lambda)}) = \mathsf{F}(x)$. For lattice-based cryptography, linear components include polynomial additions, the NTT, and the INTT. Non-linear components, such as Compress in Equation (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 in Equation (3) and Decompress in Equation (4) can be permuted. Similarly, the butterfly operations within an NTT/INTT layer can be shuffled.

To randomize data in a polynomial multiplication $c[x] = a[x] \cdot b[x]$ in a ring $\mathbb{R}_{(\rho,\eta)}$, Saarinen [Saa18] proposed computing $a'[x] = \alpha \cdot a[x]$ and $b'[x] = \beta \cdot b[x]$ where α and β are chosen randomly, uniformly, and independently from $\mathbb{Z}_{\rho}$, and multiplying $c'[x] = a'[x] \cdot b'[x]$ with $\gamma = (\alpha\,\beta)^{-1}$. Alternatively, the final step could be omitted by selecting α uniformly at random from $\mathbb{Z}_{\rho}$ and computing $\beta = \alpha^{-1}$, which implies $\gamma = 1$.

For an NTT-based multiplication, Saarinen [Saa18] suggested lowering the costs by computing $\alpha = \zeta^{i}$, $\beta = \zeta^{j}$, and $\gamma = \zeta^{-i-j}$, where i and j are chosen randomly, uniformly, and independently from $[0, \eta - 1]$. If a lookup table of the powers of ζ is available, numerous multiplications in $\mathbb{Z}_{\rho}$ can be avoided. Furthermore, unlike $\mathbb{Z}_{\rho}$, the cardinality of $[0, \eta - 1]$ is a power of two, which eliminates the need for *rejection sampling* given that *random number generators* output binary vectors. Ravi et al. [RPBC20] applied the latter technique at finer granularities: instead of generating blinding factors ζ^{i} for an entire polynomial multiplication, factors can be generated for individual NTT/INTT layers or even for individual butterflies. In its most generic form, the GS butterfly in Equation (2) is realized as in Equation (7), where blinding factors ζ^{i} and ζ^{j} will eventually cancel out.

$$\mathsf{BlindedGSButterfly}(a, b; \zeta^{k}) \triangleq \left((a+b)\,\zeta^{i}, (a-b)\,\zeta^{k+j}\right) \mod \rho. \tag{7}$$

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, i.e., the long-term secret. Not surprisingly, we also target the decapsulation.

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 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 resistance to chosen-ciphertext attacks 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 or not the execution of a keyed cryptographic algorithm fails after injecting a fault can enable key recovery. Faults of the latter type are often referred to as *safe errors* [YJ00] or *ineffective faults* [Cla07]. Below, we recapitulate three applications to lattice-based cryptography.

Bettale, Montoya, and Renault [BMR21] exploited that the secret polynomials of lattice-based schemes have relatively many coefficients that are zero—if they are drawn from a CBD or other small-error distributions. Hence, by setting these coefficients to zero and observing whether such faults are effective, many coefficients are revealed to be zero. KYBER, however, cannot be defeated, given that the CBD coefficients of the private key $\mathbf{s}$ are stored and used in the NTT domain in Algorithm 3, i.e., the transformed coefficients are virtually uniformly distributed on $[0, \rho - 1]$.

Pessl and Prokop [PP21a] skipped an instruction in the final compression step of Kyber's decryption, i.e., Line 6 in Algorithm 3, such that the observed effectiveness of the fault reveals, roughly speaking, the sign of the accumulated error Δm in Equation (6). By gathering 1000s of these inequalities, the system can be solved for the secret $(\mathbf{s}, \mathbf{e})$. Their solver is based on belief propagation because of the following two advantages over *linear programming*: large dimensions can be handled and errors in the inequalities are tolerable. Their eventual algorithm, however, was unable to exceed a 1% error rate, and attempts to increase this number were deferred as *future work*.

4.3 Ineffective Faults at Indocrypt 2021

Hermelink, Pessl, and Pöppelmann [HPP21a] solved a quasi-identical system of inequalities with a similar algorithm, but collected the inequalities using a different method: the aforementioned SCA-assisted chosen-ciphertext attacks [DTVV19, RRCB20, UXT+22] are adapted such that the message-checking oracle is realized through fault injections instead of leakage measurements. More precisely, the attacker manipulates one coefficient v_ι, where $\iota \in [0, \eta - 1]$, of the compressed ciphertext polynomial $v[x]$ of an otherwise correctly computed encapsulation by replacing Line 14 in Algorithm 2 with Eq. (8), and

the (in)ability to rectify the manipulation by faulting either input of the polynomial comparison reveals, roughly speaking, the sign of Δm in Eq. (6). Recall that the coins r used by the re-encryption are derived from the message m using a hash function, so changing a single message bit alters the entire ciphertext.

$$v^\star[x] = \mathsf{Compress}(v[x] + \lfloor \rho/4 \rfloor x^\iota). \tag{8}$$

In response to Eq. (8), the accumulated error Δm faced by the decryption in Eq. (6) increases by $\lfloor \rho/4 \rfloor$, as given in Eq. (9).

$$\Delta m^\star = \Delta m + \lfloor \rho/4 \rfloor. \tag{9}$$

From Eq. (9) and the observed correctness of the faulted decapsulation, an inequality follows in Eq. (10). The difference in strictness is due to rounding. If m_ι is correct, an attacker is able to fault coefficient v_ι in either input operand of the polynomial comparison such that the decapsulation succeeds. If m_ι is incorrect, any attempt for rectification is in vain.

$$m \neq m^\star \iff (m = 0 \wedge \Delta m > 0) \vee (m = 1 \wedge \Delta m \geq 0). \tag{10}$$

To restrict the fault model to single bit flips, which enables the use of a laser, the *Hamming distance* (HD) constraint in Equation (11) is imposed when manipulating the encapsulation. If multiple bits can reliably be flipped, the HD constraint can be removed.

$$\mathsf{HD}(\mathsf{Compress}(v), \mathsf{Compress}(v[x] + \lfloor \rho/4 \rfloor x^\iota)) = 1 \tag{11}$$

An inequality is formed as given in Eq. (12). Vector $\mathbf{x}$ consists of $\psi \triangleq 2\,\kappa\,\eta$ unknowns, each taking values in $[-\epsilon_1, \epsilon_1]$. The manipulated index ι may be identical for all collected inequalities, so that the fault-injection setup only needs to cover a single point in space and time. Considering that the coefficients of matrix $\mathbf{A}$ and vector $\mathbf{b}$ are small in absolute value, modulo operations are entirely omitted.

$$\mathbf{a}\,\mathbf{x} + b \begin{cases} \geq 0 & \text{if decapsulation fails} \\ < 0 & \text{otherwise,} \end{cases} \text{, where } b \triangleq (e_2 + \Delta v)_\iota + \begin{cases} -1 & \text{if } m_\iota = 0 \\ 0 & \text{otherwise,} \end{cases} \tag{12a}$$

$$\mathbf{x} \triangleq \begin{pmatrix} \mathsf{Poly2Vec}(\mathbf{s}[0]) \\ \vdots \\ \mathsf{Poly2Vec}(\mathbf{s}[\kappa-1]) \\ \mathsf{Poly2Vec}(\mathbf{e}[0]) \\ \vdots \\ \mathsf{Poly2Vec}(\mathbf{e}[\kappa-1]) \end{pmatrix}, \mathsf{Poly2Vec}(p) \triangleq \begin{pmatrix} p_0 \\ \vdots \\ p_{\eta-1} \end{pmatrix}, \tag{12b}$$

$$\mathbf{a}^\intercal \triangleq \begin{pmatrix} -\mathsf{Poly2VecX}(\mathbf{e}_1[0] + \Delta\mathbf{u}[0]; \iota) \\ \vdots \\ -\mathsf{Poly2VecX}(\mathbf{e}_1[\kappa-1] + \Delta\mathbf{u}[\kappa-1]; \iota) \\ \mathsf{Poly2VecX}(\mathbf{r}[0]; \iota) \\ \vdots \\ \mathsf{Poly2VecX}(\mathbf{r}[\kappa-1]; \iota) \end{pmatrix}, \mathsf{Poly2VecX}(p; \iota) \triangleq \begin{pmatrix} p_\iota \\ \vdots \\ p_0 \\ -p_{\eta-1} \\ \vdots \\ -p_{\iota+1} \end{pmatrix}. \tag{12c}$$

The solver is another variation of belief propagation, and is fed inequalities that are 100% correct, originating from software-simulated faults. Although the bit flips are assumed to be perfectly reliable, the authors mention that a few trials would suffice to cover

imperfect bit flips. 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. Therefore, the authors proposed an additional countermeasure: instead of ciphertexts c, pairs $(c, \mathsf{Hash}(c))$ 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 $\mathsf{Compress}(v; \rho, \delta_v)$.

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 $\mathsf{A} : \mathcal{S} \times \mathcal{I} \to \mathcal{O}$ 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 five parts, as shown in Figure 1.

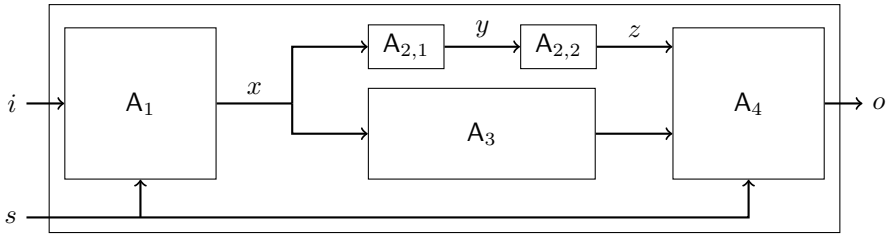

Figure 1: Decomposition of cryptographic algorithm A.

To keep the execution time of the attack within bounds, we require that cardinalities $|\mathcal{Y}|$ and $|\mathcal{Z}|$ are small. For a constant input (s, i), the attacker repeatedly faults either $\mathsf{A}_{2,1}$ or y or $\mathsf{A}_{2,2}$ or z such that $z^\star \in \mathcal{Z}$ is not constant, i.e., $z^\star$ does not follow a one-point distribution with respect to the infinite set of fault injections. Although many distributions might enable an attack, we idealize the case where $z^\star$ is uniformly distributed on $\mathcal{Z}$. In our casino analogy, this corresponds to spinning a roulette wheel, at least if we visualize $\mathcal{Z}$ through a circular representation. This analogy also emphasizes that random draws are an essential element of the attack. If for the given distribution of $z^\star$, the probability that A fails to produce the correct output o depends on the secret $s \in \mathcal{S}$, then the attacker can retrieve information on s.

Our motivation for idealizing (nearly) uniform distributions of $z^\star \in \mathcal{Z}$ is that they naturally support (i) a large attack surface and (ii) various fault models, especially when SCA countermeasures such as masking and data-randomizing blinding are deployed. Section 5.1.1 formalizes the notion that uniformly distributed faults tend to propagate as uniformly distributed faults. Section 5.1.2 gives examples of supported fault models.

5.1.1 Attack Surface

For a function that is balanced according to Definition 2, uniformly distributed faults propagate as uniformly distributed faults, as formalized in Lemma 2 and proven in Appendix C.1. If the function $A_{2,2} : \mathcal{Y} \rightarrow \mathcal{Z}$ in Figure 1 happens to be balanced, an attacker who is able fault $A_{2,1}$ or y such that the faulted value $y^\star \sim U(\mathcal{Y})$, indirectly achieves $z^\star \sim U(\mathcal{Z})$.

Definition 2 (Balanced Function)**.** Let $\mathsf{F} : \mathcal{A} \rightarrow \mathcal{C}$ be a function. If it holds $\forall c \in \mathcal{C}$ that $|\{a \in \mathcal{A} \mid \mathsf{F}(a) = c\}| = |\mathcal{A}|/|\mathcal{C}|$, then F is balanced. Similarly, for a function $\mathsf{F} : \mathcal{A} \times \mathcal{B} \rightarrow \mathcal{C}$, if it holds $\forall (b, c) \in \mathcal{B} \times \mathcal{C}$ that $|\{a \in \mathcal{A} \mid \mathsf{F}(a, b) = c\}| = |\mathcal{A}|/|\mathcal{C}|$, then F is balanced with respect to input $a \in \mathcal{A}$.

Lemma 2 (Fault Propagation for Balanced Functions)**.** *Let $\mathsf{F} : \mathcal{A} \rightarrow \mathcal{C}$ be a balanced function, as formalized in Definition 2. If $a \sim U(\mathcal{A})$, then $c \sim U(\mathcal{C})$. Similarly, for a function $\mathsf{F} : \mathcal{A} \times \mathcal{B} \rightarrow \mathcal{C}$ that is balanced with respect to input $a \in \mathcal{A}$, if $a \sim U(\mathcal{A})$ is independent of $b \in \mathcal{B}$, then $c \sim U(\mathcal{C})$.*

Fortunately for the attacker, balanced functions are frequently used in cryptography. Bijections are a trivial example. Addition in a finite ring and multiplication 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 may still enable an attack in practice.

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

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

5.1.2 Fault Models

Examples 1 to 4 demonstrate that the ideal distribution, $z^\star \sim U(\mathcal{Z})$, can be achieved for various fault models. Despite assuming that the attacker faults either $A_{2,2}$ or z, balanced fault-propagation properties according to Section 5.1.1 may extend the attack surface to $A_{2,1}$ and y. Again, note that a uniform distribution is merely the ideal case; other distributions may enable an attack as well. Regarding our distinction in Section 3.2, remark that roulette attacks are facilitated by blinding methods that randomize data but counteracted by blinding methods that randomize time.

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

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

Example 3 (Instruction Skips and Corruptions)**.** Let $\mathsf{A}_{2,2} : \mathcal{Y} \rightarrow \mathcal{Z}$ be realized through a masked software implementation. Without loss of generality, assume that an instruction

in the first share function, $\mathsf{A}_{2,2}^{(1)}$, is either skipped or corrupted such that the faulty output share $(z^{(1)})^\star$ is independent of the correct output share $z^{(1)}$. Hence, $z^\star = (z^{(1)})^\star + z^{(2)} + \cdots + z^{(\lambda)}$ is again uniformly distributed on $\mathcal{Z}$.

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

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 $o^\star$	Known	Unknown	Unknown	Unknown
Input i	Constant i	Constant i	$i \leftarrow U(\mathcal{I})$	Constant i
Correct intermediate z	Constant z	Constant z	$z \sim U(\mathcal{Z})$	Constant z
Faulty intermediate $z^\star$	Any	Any	$z^\star \not\sim U(\mathcal{Z})$	Any, $z^\star \sim U(\mathcal{Z})$
Fault intensity	Constant	Variable	Constant	Constant
Masking	Nuisance	Nuisance	Nuisance	Facilitator
Duplication	Game over	Don't care	Don't care	Don't care

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 paper [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.

5.2.1 Attack Surface

The generic variable $z \in \mathcal{Z}$ in Figure 1 is instantiated with a compressed ciphertext coefficient $v \in \{0,1\}^{\delta_v}$ 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 $v^\star$ is uniformly distributed on $\{0,1\}^{\delta_v}$, then the probability of a successful decapsulation is approximately 0 if $m \neq m^\star$ 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 attack of Hermelink et al. [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 Figure 2, the function $A_2 \triangleq A_{2,2} \circ A_{2,1}$ that produces a coefficient $v \in \{0,1\}^{\delta_v}$ comprises one GS butterfly in the last layer of an INTT, the generation of one CBD sample, the decompression of one message bit, one modular addition, and one compression. Moreover, by faulting any of these building blocks, the countermeasure of Hermelink et al. [HPP21a] to store $(c, \mathsf{Hash}(c))$ in RAM is bypassed.

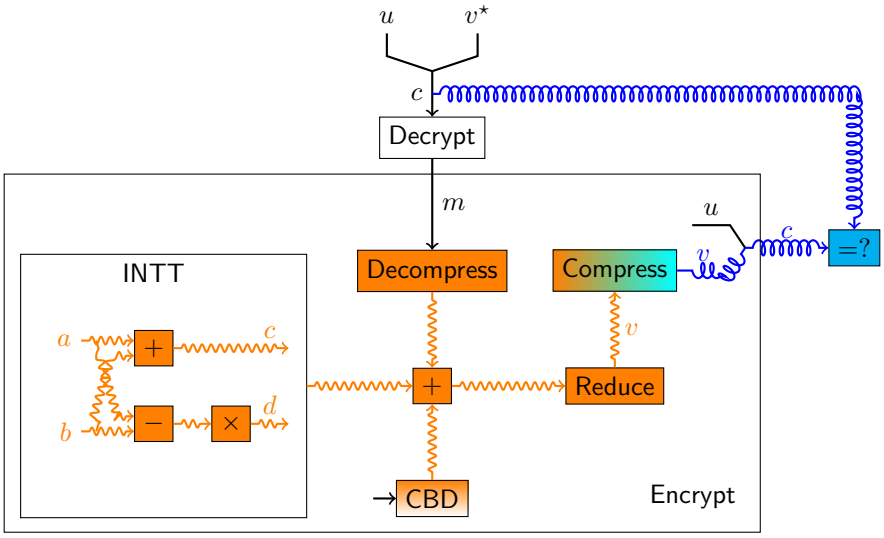

Figure 2: The attack surface of the IndoCrypt paper [HPP21a] is colored blue; our extension is colored orange.

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 Equation (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 is uniformly distributed on $[0, \rho - 1]$, then the compressed coefficient 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.

5.2.2 Optional Hamming-Distance Constraint

The sole purpose of the Hamming distance constraint in Equation (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 extend Equations 8 to 10. As a starting point, we summarize the behavior of Compress in Equation (3) and Decompress in Equation (4). For KYBER512 and KYBER768, where $\delta_v = 4$, our summary is contained in the first five columns of Table 3. For brevity, we do not discuss KYBER1024, where $\delta_v = 5$, but identical conclusions can be drawn from Table 5 in Appendix A.

Table 3: Properties of the compressed ciphertext coefficients $v \in [0, 2^\delta - 1]$ where $\delta = 4$. The first and last elements of each bin are defined by Compress in Equation 3. The bin centers are defined by Decompress in Equation 4.

Original					Manipulated			
Bin	Size	First	Last	Center	Bin	Fault	HD	$\Delta m^\star$
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		1353	1560	1456	11			
8	208	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			

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. (8) is to add $\lfloor \rho/4 \rfloor = 832 = 4 \cdot 208$ to the uncompressed coefficient, 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 $3225+832 \mod 3329 = 728$, and thus not to the first element of bin 4. In absence of the HD constraint in Eq. (11), the decryption would face an accumulated error $\Delta m^\star = \Delta m + 632$, which significantly undershoots the desired effect $\Delta m^\star = \Delta m + 832$ in Eq. (9). An easy fix is to replace Eq. (8) by a direct manipulation of the compressed coefficient, as given in Eq. (13).

$$v^\star = v + 2^{\delta_v - 2} \mod 2^{\delta_v}. \tag{13}$$

Furthermore, in cases where the HD is 2 instead of 1, the accumulated error Δm happens to be increased by 833 instead of 832. The required extension of Eqs. (9) and (10) is given in Eq. (14).

$$\Delta m^\star = \Delta m + 832 \implies m \neq m^\star \iff (m = 0 \wedge \Delta m \geq 1) \vee (m = 1 \wedge \Delta m \geq 0), \tag{14a}$$

$$\Delta m^\star = \Delta m + 833 \implies m \neq m^\star \iff (m = 0 \wedge \Delta m \geq 0) \vee (m = 1 \wedge \Delta m \geq -1). \tag{14b}$$

Similarly, Eq. (12a) is extended in Eq. (15).

$$b \triangleq (e_2 + \Delta v)_\iota + \begin{cases} -1 & \text{if } m_\iota = 0 \text{ and } \Delta m^\star = \Delta m + 832 \\ 1 & \text{if } m_\iota = 1 \text{ and } \Delta m^\star = \Delta m + 833 \\ 0 & \text{otherwise.} \end{cases} \tag{15}$$

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. 2, where most building blocks come 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	$c_1^\star$	$d_1^\star$	$c_2^\star$	$d_2^\star$	Proof
1	usub16 t_1, a, b	✓	$\sim U(\mathbb{Z}_\rho)$	✓	$\sim U(\mathbb{Z}_\rho)$	Eq. (26)
2	uadd16 a, a, b	$\sim U(\mathbb{Z}_\rho)$	✓	$\sim U(\mathbb{Z}_\rho)$	✓	Eq. (23)
3	smulbb b, t_1, τ	✓	$\sim U(\mathbb{Z}_\rho)$	✓	✓	Eq. (24)
4	smultb t_1, t_1, τ	✓	✓	✓	$\sim U(\mathbb{Z}_\rho)$	Eq. (25)
5.1	smulbb 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	smulbb 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, asr#16	✓	$\sim U(\mathbb{Z}_\rho)$	✓	✓	Eq. (27)

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 $d^\star$ might not exist. However, we took an empirical approach by measuring the distribution of $d^\star$ 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 Equation (7) in the final INTT layer. As formalized in Equation (16), if the attacker flips an arbitrary set of bits in multiplicand $(a + b)$, then the faulted butterfly output $c^\star$ is uniformly distributed on a subset of $\mathbb{Z}_\rho$ 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.

$$(a+b)^\star \triangleq (a+b) \oplus e \implies \Delta c \triangleq c^\star - c = \left(\sum_{n=0}^{\lfloor \log_2(\rho) \rfloor} e[n](-1)^{(a+b)[n]} 2^n \right) \zeta^i. \tag{16}$$

Similarly, bit flips in multiplicand $(a-b)$ cause butterfly output d to be uniformly distributed on a subset of η elements in $\mathbb{Z}_\rho$. It also possible to flip bits of either a or b, but then more injections must be performed considering that c and d are simultaneously faulted.

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. With Pessl as a common author, it is also the first third-party validation. 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, the public key, 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 times from previous solvers can be attributed to a single culprit, i.e., Eq. (17). Belief-propagation algorithms maintain a *probability mass function* (PMF) for each out of ψ unknowns $X[j]$, to be initialized with the CBD, and in each iteration, these PMFs are updated based on the probabilities $p[i,j,k]$ in Eq. (17), until they converge to one-point distributions. After the first iteration, the PMFs of $X[j]$ do not have a special shape anymore, so the PMF of the sum of $\psi - 1$ random variables is computed through general means: linear (non-circular) convolutions via the *fast Fourier transform* (FFT). The previous solvers [PP21b, HPP21b] use *binary trees* to improve the reuse of intermediate variables, yet with $\omega\,\psi$ FFTs and $\omega\,\psi$ inverse FFTs per iteration, the load remains heavy.

$$\begin{matrix} \forall i \in [0, \omega-1], \\ \forall j \in [0, \psi-1], \\ \forall k \in [0, 2\,\epsilon_1], \end{matrix} \quad p[i,j,k] = \Pr\left(\mathbf{A}[i,j]\,(k-\epsilon_1) + \left(\sum_{j' \in [0,\psi-1] \setminus \{j\}} \mathbf{A}[i,j']\,X[j'] \right) + b[i] \geq 0 \right). \tag{17}$$

We accelerate Eq. (17) 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. (18) 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.

$$p[i,j,k] \approx \mathsf{F}_{\text{norm}}\left(\frac{\frac{1}{2} + \mathbf{A}[i,j]\,(k-\epsilon_1) + \sum_{j' \in [0,\psi-1]\setminus\{j\}} \mathbf{A}[i,j']\,\mathrm{E}\big[X[j']\big]}{\sqrt{\sum_{j' \in [0,\psi-1]\setminus\{j\}} \mathbf{A}[i,j']^2\ \mathrm{Var}\big[X[j']\big]}}\right). \tag{18}$$

Instead of the reported 15 minutes, a single-threaded 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 entire solver is represented in Algorithm 9, including changes to increase the error tolerance. Whilst observed decapsulation successes are correctly classified with quasi 100% certainty, observed decapsulation failures are only correct up to a probability that is estimated in Lines 4 to 7. In Line 14, this information is taken into account.

Algorithm 9 Solver

Input: Matrix $\mathbf{A}$
Input: Vector b
Input: Decapsulation failures $r \in \{0,1\}^{\omega}$
Output: Solution $x_{\text{guess}} \in [-\epsilon_1, \epsilon_1]^{\psi}$

1: **for** $j \in [0, \psi-1]$ **do**
2: **for** $k \in [0, 2\epsilon_1]$ **do**
3: $p_{\text{unknowns}}[j,k] \leftarrow \mathsf{f}_{\text{bino}}(k; 2\,\epsilon_1, 1/2)$ ▷ Proof in Eq. (28).
4: $p_{\text{fail,expected}} \leftarrow \frac{1}{\omega}\sum_{i=0}^{\omega-1} \Pr\big(\sum_{j=0}^{\omega-1} A[i,j]\,x[j] + b[i] \geq 0\big)$ ▷ CLT approximation
5: $p_{\text{fail,measured}} \leftarrow \frac{1}{\omega}\sum_{i=0}^{\omega-1} r[i]$
6: $p_{\text{fail,correct}} \leftarrow \min(p_{\text{fail,expected}}/p_{\text{fail,measured}}, 1)$
7: $p_{\text{fail}} \leftarrow r \cdot p_{\text{fail,correct}}$
8: $x_{\text{guess}} \leftarrow \begin{pmatrix}0 & 0 & \cdots & 0\end{pmatrix}$
9: **while** StopCriterionFails$(\mathbf{A}, b, r, x_{\text{guess}}, \cdots)$ **do**
10: **for** $j \in [0, \psi-1]$ **do**
11: **for** $k \in [0, 2\,\epsilon_1]$ **do**
12: **for** $i \in [0, \omega-1]$ **do**
13: Compute $p[i,j,k]$ using Eq. (18) ▷ CLT approximation.
14: $p[i,j,k] \leftarrow p[i,j,k]\,p_{\text{fail}}[i] + (1-p[i,j,k])(1-p_{\text{fail}}[i])$
15: $p[i,j,k] \leftarrow \max(p[i,j,k], 10^{-5})$
16: $p'_{\text{unknowns}}[j,k] = p_{\text{unknowns}}[j,k]\,\prod_{i=0}^{\omega-1} p[i,j,k]$ ▷ Sum of logarithms
17: **for** $k \in [0, 2\,\epsilon_1]$ **do**
18: $p'_{\text{unknowns}}[j,k] \leftarrow p'_{\text{unknowns}}[j,k] / \sum_{k'=0}^{2\,\epsilon_1} p'_{\text{unknowns}}[j,k']$ ▷ Normalize
19: $x_{\text{guess}}[j] \leftarrow -\epsilon_1 + \arg\max_{k \in [0,2\,\epsilon_1]} p'_{\text{unknowns}}[j,k]$
20: $p_{\text{unknowns}} \leftarrow p'_{\text{unknowns}}$

Regarding the CBD in Line 3, we point out that the PMF of $E \triangleq E_1 - E_2$ where $E_1, E_2 \sim B(\epsilon, 1/2)$ can simply be evaluated as $\mathsf{f}_{\text{bino}}(\epsilon - i; 2\epsilon, 1/2)$, as proven in Appendix C.5. Not equally compact, Hermelink [HPP21b] loops over all pairs $(e_1, e_2) \in [0,\epsilon]^2$. As the probabilities $p[i,j,k]$ may be small, the product in Line 16 is realized through a sum of logarithms to avoid underflow. Line 15 ensures that the logarithms do not receive inputs

close to zero. The *stop criterion* in Line 9 is met if a maximum of 16 iterations is reached, or if an estimated fitness of x_{guess} obtained by filling in the inequalities does not improve anymore.

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. 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.

6.3 Experiments with Software-Simulated Faults

We perform three experiments where faults are simulated in software. Success is quantified by measuring the probability that the coefficients of x_{guess} are correct, as a function of the provided number of inequalities ω. Each probability is averaged over either 5 or 10 systems of inequalities, which correspond to different key pairs. 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. 3. 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.

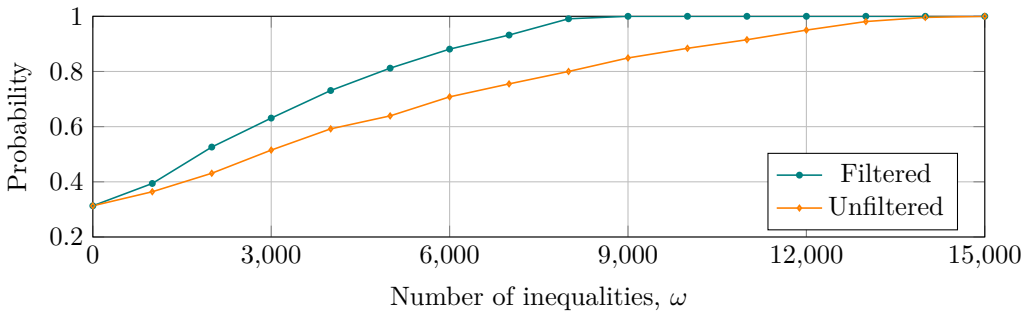

Figure 3: Filtered and unfiltered inequalities for Kyber512.

For our second experiments, all three security levels of Kyber are compared in Fig. 4. 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 ($7 > 5$).

For our third experiment, inequalities are corrupted. In line with the working principles of the attack, inequalities corresponding to a decapsulation failure are untouched, whereas decapsulation successes are turned into decapsulation failures with probability p_{s2f}. Figure 5 shows that even with $p_{\text{s2f}} = 50\%$, 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.

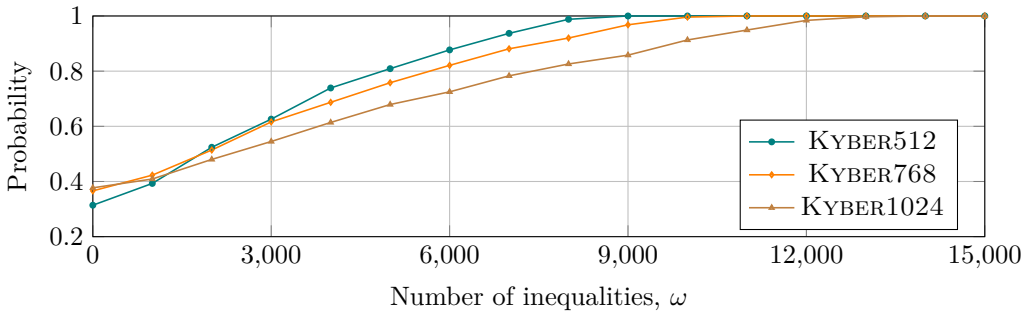

Figure 4: All security levels.

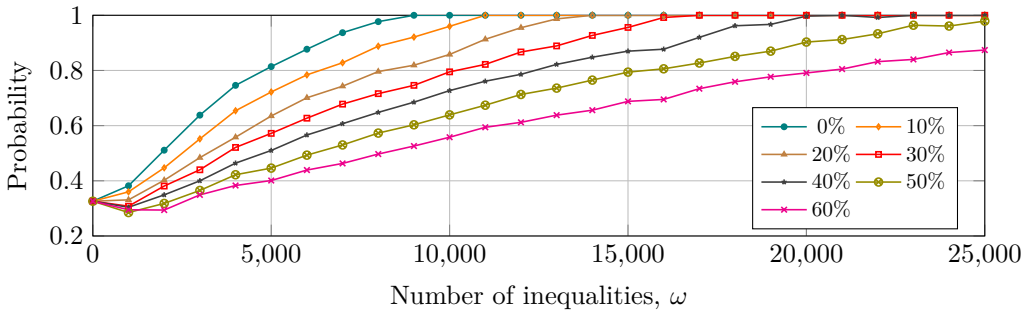

Figure 5: Error tolerance for Kyber512. Given ω otherwise correct inequalities, decapsulation successes are turned into decapsulation failures with probability $p_{\text{s2f}} \in [0, 0.6]$.

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, and (iii) the implementation of Heinz et al. [HKL+22] for having an unresolved masking issue 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. [Inc] 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. 6 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. 5, 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.

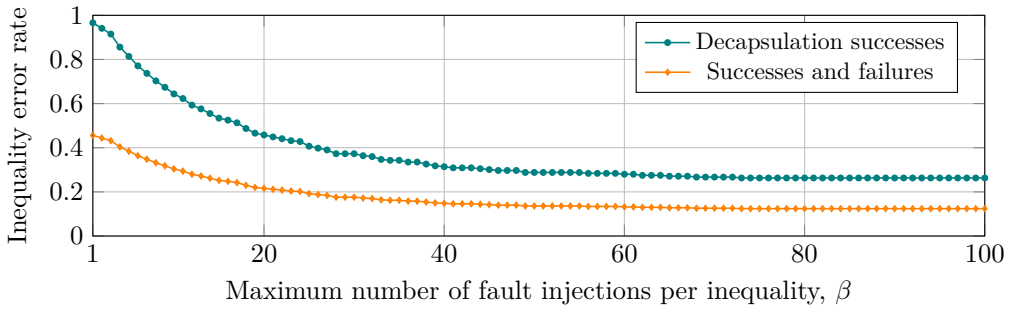

Figure 6: Inequality error rates obtained by clock glitching an ARM Cortex-M4 using a ChipWhisperer board.

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. Popular masking techniques against SCA that originally favoured the defender now favor the attacker. Furthermore, more building blocks can be attacked, thereby increasing expenses for the defender. Finally, 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. In light of the above, the design of effective yet affordable countermeasures against our roulette attacks is a first suggestion for follow-up work.

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 and round-3 finalist. 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 $c_m^\star$ is uniformly distributed on $\{0,1\}^\tau$. Furthermore, $c_m^\star$ 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.

Acknowledgements

The author thanks Santos Merino Del Pozo for proofreading an earlier version of this manuscript.

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, 2020.

[ABD+20] Roberto Avanzi, Joppe Bos, Léo Ducas, Eike Kiltz, Tancrède 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.

[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, 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 DAnvers, 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 Guénaël Renault. Safe-error analysis of post-quantum cryptography mechanisms. Cryptology ePrint Archive, Report 2021/1339, October 2021. https://ia.cr/2021/1339.

[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, *Cryptographic Hardware and Embedded Systems - CHES 2007, 9th International Workshop, Vienna, Austria, September 10-13, 2007, Proceedings*, volume 4727 of *Lecture Notes in Computer Science*, pages 181–194. Springer, 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, 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.

[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 *Progress in Cryptology - INDOCRYPT 2021*, Lecture Notes in Computer Science, page 25. Springer, 2021. https://ia.cr/2021/1222.

[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.

[Inc] NewAE Technology Inc. Chipwhisperer. https://www.newae.com/chipwhisperer. Accessed: April 8, 2022.

[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.

[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.

[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, 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, 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.

[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, *Security, Privacy, and Applied Cryptography Engineering - 10th International Conference, SPACE 2020, Kolkata, India, December 17-21, 2020, Proceedings*, volume 12586 of *Lecture Notes in Computer Science*, pages 123–146. Springer, 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, *Constructive Side-Channel Analysis and Secure Design - 10th International Workshop, COSADE 2019, Darmstadt, Germany, April 3-5, 2019, Proceedings*, volume 11421 of *Lecture Notes in Computer Science*, pages 232–250. Springer, 2019.

[RRCB20] Prasanna Ravi, Sujoy Sinha Roy, Anupam Chattopadhyay, and Shivam Bhasin. Generic side-channel attacks on CCA-secure lattice-based PKE and KEMs. *IACR Transactions on Cryptographic Hardware and Embedded Systems (TCHES)*, 2020(3):307–335, 2020.

[Saa18] Markku-Juhani O. Saarinen. Arithmetic coding and blinding countermeasures for lattice signatures. *Journal of Cryptographic Engineering*, 8(1):71–84, 2018.

[SH07] Jörn-Marc Schmidt and Michael Hutter. Optical and EM fault-attacks on CRT-based RSA. In *15th Austrian Workhop 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+22] 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.

[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*, Lecture Notes in Computer Science. Springer, 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, 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	$\Delta m^\star$
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		1509	1612	1560	23			
16	104	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. 1 is instantiated with an uncompressed message coefficient $m \in [0, \rho-1]$. Although practically any distribution of its faulted counterpart $m^\star$ enables the attack, at least if the distribution is known to the attacker, we again idealize the case where $m^\star$ is uniformly distributed on $[0, \rho-1]$. Leveraging fault propagation, the attack surface consists of $\mathsf{Decompress}(v; \rho, \delta_v)$, 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 $m^\star$. Given that primes ρ are odd, the final decryption step, i.e., $m^\star \leftarrow \mathsf{Compress}(m^\star; \rho, 1)$ as defined in Eq. (5), is inherently biased. As illustrated in Fig. 7 for $\rho = 7$, the compression function maps $\lfloor \rho/2 \rfloor = 3$ coefficients in $[0, \rho-1]$ to $m^\star = 0$, whereas $\lceil \rho/2 \rceil = 4$ coefficients map to $m^\star = 1$.

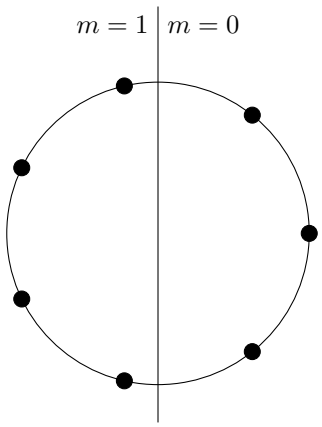

Figure 7: Message coefficients m before and after compression according to Eq. (5) where prime $\rho = 7$.

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 $\mathsf{F} : \mathcal{A} \to \mathcal{C}$ of Lemma 2 is proven in Eq. (19); the case $\mathsf{F} : \mathcal{A} \times \mathcal{B} \to \mathcal{C}$ is proven in Eq. (20).

$$\Pr(c) = \sum_{\substack{a \in \mathcal{A} \text{ s.t.} \\ \mathsf{F}(a) = c}} \Pr(a) = \frac{|\mathcal{A}|}{|\mathcal{C}|} \cdot \frac{1}{|\mathcal{A}|} = \frac{1}{|\mathcal{C}|}. \tag{19}$$

$$\begin{aligned}\Pr(c) = \sum_{\substack{(a,b)\in\mathcal{A}\times\mathcal{B}\\ \text{s.t. } \mathsf{F}(a,b)=c}} \Pr(a\wedge b) &= \sum_{b\in\mathcal{B}} \Pr(b) \sum_{\substack{a\in\mathcal{A} \text{ s.t.}\\ \mathsf{F}(a,b)=c}} \Pr(a) \\ &= \frac{|\mathcal{A}|}{|\mathcal{C}|}\cdot\frac{1}{|\mathcal{A}|}\cdot\sum_{b\in\mathcal{B}}\Pr(b) = \frac{1}{|\mathcal{C}|}.\end{aligned} \tag{20}$$

C.2 Lemma 3

Balancedness with respect to input $a \in \mathcal{R}$ in Lemma 3 is proven in Eq. (21) and follows from the property that each element in a ring has an *additive inverse*. Balancedness with respect to input $b \in \mathcal{R}$ is proven in an identical manner.

$$\forall (b,c) \in \mathcal{R}^2, |\{a \in \mathcal{R} \mid a+b=c\}| = |\{c-b\}| = 1. \tag{21}$$

C.3 Lemma 4

Balancedness with respect to input $a \in \mathcal{F}$ in Lemma 4 is proven in Eq. (22) and follows from the property that each element $b \neq 0$ in a field has an *multiplicative inverse*.

$$\forall (b,c) \in \mathcal{F}^2, |\{a \in \mathcal{F} \mid a\cdot b=c\}| = |\{c\cdot b^{-1}\}| = 1. \tag{22}$$

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 $\mathbb{Z}_\rho^\eta$.

The proofs for instructions 2 to 4 in Table 4 are particularly straightforward and given in Eqs. (23) to (25) 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.

$$\begin{cases} \left(c_1^{(1)}, c_2^{(1)}\right)^\star = \left(a_1^{(1)}, a_2^{(1)}\right), \\ \left(c_1^{(2)}, c_2^{(2)}\right) = \left(a_1^{(2)}+b_1^{(2)}, a_2^{(2)}+b_2^{(2)}\right), \\ \quad\vdots \\ \left(c_1^{(\lambda)}, c_2^{(\lambda)}\right) = \left(a_1^{(\lambda)}+b_1^{(\lambda)}, a_2^{(\lambda)}+b_2^{(\lambda)}\right), \end{cases} \implies \left(c_1, c_2\right)^\star = (c_1,c_2) - \left(b_1^{(1)}, b_2^{(1)}\right) \sim U(\mathbb{Z}_\rho^2). \tag{23}$$

$$\begin{cases} \left(d_1^{(1)}\right)^\star = b_1^{(1)}, \\ d_1^{(2)} = \left(a_1^{(2)} - b_1^{(2)}\right)\tau, \\ \quad\vdots \\ d_1^{(\lambda)} = \left(a_1^{(\lambda)} - b_1^{(\lambda)}\right)\tau, \end{cases} \implies d_1^\star = d_1 + b_1^{(1)}(\tau+1) - a_1^{(1)}\tau \sim U(\mathbb{Z}_\rho). \tag{24}$$

$$\begin{cases} \big(d_2^{(1)}\big)^\star = a_2^{(1)} - b_2^{(1)}, \\ d_2^{(2)} = \big(a_2^{(2)} - b_2^{(2)}\big)\tau, \\ \quad\vdots \\ d_2^{(\lambda)} = \big(a_2^{(\lambda)} - b_2^{(\lambda)}\big)\tau \end{cases} \implies d_2^\star = d_2 + \big(a_2^{(1)} - b_2^{(1)}\big)(1-\tau) \sim U(\mathbb{Z}_\rho). \tag{25}$$

For instruction 1 in Table 4, the faulted output coefficients $(d_1, d_2)^\star$ are determined by an uninitialized temporary variable t_1, as formalized in Eq. (26). 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.

$$\begin{cases} \big(d_1^{(1)}, d_2^{(1)}\big)^\star = \big(t_1[15:0], t_1[31:16]\big)\,\tau, \\ \big(d_1^{(2)}, d_2^{(2)}\big) = \big(a_1^{(2)} - b_1^{(2)}, a_2^{(2)} - b_2^{(2)}\big)\,\tau, \\ \quad\vdots \\ \big(d_1^{(\lambda)}, d_2^{(\lambda)}\big) = \big(a_1^{(\lambda)} - b_1^{(\lambda)}, a_2^{(\lambda)} - b_2^{(\lambda)}\big)\,\tau, \end{cases} \quad \begin{array}{r} \text{where } t_1 \text{ and} \\ \big(a_1^{(1)}, b_1^{(1)}, a_2^{(1)}, b_2^{(1)}\big) \\ \text{are independent,} \end{array} \tag{26a}$$

$$\implies \begin{array}{r} \big(d_1, d_2\big)^\star = (d_1, d_2) + \big(t_1[15:0] - a_1^{(1)} + b_1^{(1)}, \\ t_1[31:16] - a_2^{(1)} + b_2^{(1)}\big)\,\tau \sim U(\mathbb{Z}_\rho^2) \end{array} \tag{26b}$$

For instruction 7 in Table 4, the faulty output coefficient $d_1^\star$ is uniformly distributed on $\mathbb{Z}_\rho$ in theory, but not necessarily in practice. The output of the function 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. (13), but are hard to analyze from a mathematical perspective and not further addressed here.

$$\begin{cases} \big(d_1^{(1)}\big)^\star = \mathsf{M}\Big(\big(a_2^{(1)} - b_2^{(1)}\big)\tau\Big), \\ d_1^{(2)} = \big(a_1^{(2)} - b_1^{(2)}\big)\tau, \\ \quad\vdots \\ d_1^{(\lambda)} = \big(a_1^{(\lambda)} - b_1^{(\lambda)}\big)\tau \end{cases} \implies \begin{array}{r} d_1^\star = d_1 + \mathsf{M}\Big(\big(a_2^{(1)} - b_2^{(1)}\big)\tau\Big) \\ -\big(a_1^{(1)} - b_1^{(1)}\big)\tau \sim U(\mathbb{Z}_\rho). \end{array} \tag{27}$$

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. (28). Vandermonde's identity is used in Eq. (28c) to dispose of the summation operator.

$$\Pr(X = i) = \Pr(X = |i|) = \sum_{j=0}^{\epsilon-|i|} \Pr(X_2 = j)\Pr(X_1 = |i| + j) \tag{28a}$$

$$= \sum_{j=0}^{\epsilon-|i|} \mathsf{f}_{\text{bino}}(j; \epsilon, 1/2)\,\mathsf{f}_{\text{bino}}(|i| + j; \epsilon, 1/2) = \sum_{j=0}^{\epsilon-|i|} \binom{\epsilon}{j}\left(\frac{1}{2}\right)^\epsilon \binom{\epsilon}{|i|+j}\left(\frac{1}{2}\right)^\epsilon \tag{28b}$$

$$= \left(\frac{1}{2}\right)^{2\epsilon} \sum_{j=0}^{\epsilon-|i|} \binom{\epsilon}{j} \binom{\epsilon}{\epsilon-|i|-j} = \left(\frac{1}{2}\right)^{2\epsilon} \binom{2\epsilon}{\epsilon-|i|} = \mathsf{f}_{\text{bino}}(\epsilon - i; 2\epsilon, 1/2). \tag{28c}$$