

Default Correct: A New Fault Surface in the Comparison Booleanisation of Kyber-KEM

Anirudh Jaiswal¹, Abhilash Kumar Das² and Dhiman Saha³

[de.ci.phe.red](#) Lab

Department of Computer Science and Engineering

Indian Institute of Technology Bhilai, India

{¹[anirudhjaiswal](#), ²[abhilashkd](#), ³[dhiman](#)}@iitbhilai.ac.in

Abstract. The Fujisaki–Okamoto (FO) transform protects Kyber-KEM against chosen-ciphertext attacks and is based on a ciphertext comparison step. This comparison is usually treated as a single atomic check. However, in every mainstream implementation it is a short *pipeline of independently faultable stages*. It comprises a byte-wise mismatch accumulation, a two’s-complement *Booleanisation*, and a conditional move (`cmov`). In this work, we expose a previously unexamined stage of this pipeline, the Booleanisation, as a new attack surface. Our research reveals that a single sub-instruction clock glitch forces the mismatch flag *fail* to 0, so decapsulation accepts *any* ciphertext. Every prior fault attack on this comparison targets the equality test or the conditional move. So does the only countermeasure proposed for it: the *default-fail cmov* ordering of Xagawa *et al.* Our surface sits upstream of that move, so it escapes both. We instantiate this **Default Correct** fault on an ARM Cortex-M4 (ChipWhisperer-Lite) and demonstrate it across the *pqm4*, PQClean, and reference implementations of Kyber-512/768/1024 at all optimisation levels (`-O0-O3`). Crucially, every one of these implementations already carries Xagawa’s default-fail ordering, yet the fault succeeds on all of them: the surface is therefore not merely new but structurally beyond the reach of the state-of-the-art defence for this comparison. The Booleanisation is thus a single point of failure common to all three code bases. Leveraging the accept-everything behaviour, we use the standard plaintext-checking oracle and recover the full secret key of Kyber-512, Kyber-768, and Kyber-1024 in a single on-device run each. Finally, we analyse why defences that protect only the `cmov` or a single *fail* flag cannot address a fault that targets an upstream stage of the comparison, and we propose a producer-side countermeasure: an integrity-checked Booleanisation that computes the same flag while folding in an execution counter, so that any partial instruction skip is detected, at negligible overhead.

Keywords: Kyber-KEM · ML-KEM · Fujisaki–Okamoto transform · Ciphertext comparison · Booleanisation · Fault injection · Instruction skip · Countermeasure bypass · Post-quantum cryptography

1 Introduction

The development of quantum computing poses a serious threat to classical public key cryptography. Since the introduction of Shor’s algorithm [Sho94b], it has been known that widely deployed schemes such as RSA and ECC can be broken efficiently on a large-scale quantum computer [Sho94a]. In response, NIST initiated the standardization process for post-quantum cryptography [Nat16]. Among the proposed candidates, lattice-based constructions have emerged as leading solutions due to their strong security guarantees and efficient performance [AAC⁺22]. In particular, Kyber is a lattice-based key encapsulation mechanism (KEM), which is now standardized.

At its lattice core, Kyber provides only chosen-plaintext (IND-CPA) security; the stronger chosen-ciphertext (IND-CCA) security demanded of a deployed KEM is obtained by wrapping that core in the Fujisaki–Okamoto (FO) transform [FO99, Den03, FO13]. The mechanism is simple but load-bearing: to decapsulate, the receiver decrypts the incoming ciphertext, deterministically re-encrypts the recovered message, and compares the recomputed ciphertext against the one it received. The true session key is released only if the two match; otherwise an implicit-rejection value is returned.

The single equality check in FO is what separates a well-formed query from a malformed one. It is therefore a central operation on which Kyber’s CCA security rests thereby making its runtime integrity security-critical. While the theoretical security of lattice KEMs is well studied, their practical security depends heavily on implementation aspects [RCDB22]: physical attacks such as side-channel analysis and fault injection can break an implementation even when the underlying scheme is sound [PP19, PP21, BDH⁺21]. Fault injection [BS97, BDL01, SMR09, ESH⁺11, HHS⁺11] is especially dangerous here, because forcing the comparison to accept a ciphertext it should reject, collapses CCA security back to CPA and turns decapsulation into a key-recovery oracle. Yet the comparison only *looks* atomic. In every mainstream implementation it is a short pipeline of stages: a byte-wise accumulation of ciphertext mismatches, a two’s-complement *Booleanisation* that reduces the accumulator to a single accept/reject flag, and a conditional move (`cmov`) that releases the key on that flag. So far, every fault attack on this comparison has engaged only its *final* stage: the conditional move, or the equality test that feeds it. The same is true of the sole countermeasure proposed for it, the *default-fail cmov* ordering of Xagawa *et al.* [XIU⁺21]. That defence pre-loads the output with the rejection value, so a skipped `cmov` leaks nothing.

In this work we expose a previously unexamined stage of that pipeline as a new fault surface: the Booleanisation itself. A single sub-instruction glitch on the two’s-complement sequence forces the mismatch flag *fail* to 0, so decapsulation accepts any ciphertext. Because this stage sits *upstream* of the `cmov`, it is invisible to any defence scoped to the conditional move. The `cmov` executes exactly as designed and faithfully releases the key on a flag that was corrupted before it ever arrived. The surface is therefore not merely new but structurally outside the reach of the state-of-the-art defence. Every implementation we attack already carries Xagawa’s default-fail ordering, yet the fault succeeds on all of them. Hardening a single stage of a multi-stage check, we conclude, buys only a false sense of security.

The faultable Booleanisation could be a side-effect of one implementation or a property of how the FO comparison is implemented in practice. Our evidence points to the implementation pattern, not to the FO transform. The ciphertext check is reduced to a single Booleanised flag. This flag is *produced* in one routine and *consumed* in another. A defence that protects the consumer cannot constrain the producer. The default-fail principle that hardens the `cmov` was never applied to the producer, whose natural reductions default to *accept* when an instruction is skipped. This pattern is not a coding accident. It is the standard constant-time idiom for the FO comparison, and all three mainstream code bases we study share it. The pattern also survives compilation. Across all 36 configurations, the Booleanisation reduces to the same `negs/sbc.w` (or `rsbs/sbc.w`) sequence. Only the register allocation and the absolute position differ. As a result, the glitch-intensity parameters stay constant and only the cycle offset is retuned per build (Section 6.1). The exposure is therefore structural to the single-flag design and not tied to any particular binary. The claim applies to comparisons that reduce the decision to a single unguarded flag, and not to the FO transform itself. Designs that diffuse the decision, such as infective computation, do not share it. Neither do designs that verify the producer, such as redundancy or the integrity-checked Booleanisation of Section 7. Masked implementations are a separate case. Masking targets side channels, and its effect on this fault depends on

whether the shares of the flag are recombined into a single unmasked value before the `cmov`. If they are, the single-flag pattern reappears at the recombination. The *mkm4* [HKL⁺22] and *pqm4_masked* [SIM22, BC22] libraries are such cases: they retain the producer and consumer stages, but their producers realise the comparison predicate through a masked share-wise comparison and a distinct recombination step rather than the two’s-complement idiom, so our fault parameters do not transfer and the effect of a skip at the recombination is open. If a masked selector consumes the shares directly, no single flag exists and our claim does not apply. The same single-flag idiom is the generic constant-time `verify()` used by FO-based KEM implementations beyond Kyber. We therefore expect the surface to extend to them, and we leave its confirmation to future work.

Related work. Physical attacks recover secrets from lattice KEMs even when the underlying scheme is secure, along two lines: passive leakage, namely power and electromagnetic analysis [EFGT17, Pes16] and cache-timing [GBHLY16], and active fault injection [BP18]. We concentrate on the latter as it applies to the FO decapsulation of Kyber.

The most direct fault target is the decapsulation pipeline itself. Ravi *et al.* [RBRC20] identified the message decoder as a common weak point across lattice schemes and mounted electromagnetic fault attacks on *pqm4* [KRSS19]; more pointedly, skipping the ciphertext equality check turns decapsulation into an accept-everything primitive [XIU⁺21], which drives plaintext-checking and decryption-failure oracles for full key recovery [UXT⁺22, RBRB⁺18, GJN19, RRCB20, QCZ⁺21]. Adjacent stages have also been faulted: Pessl and Prokop [PP21] exploit ineffective faults in decoding, Hermelink *et al.* [HPP21] combine a ciphertext bit-flip with fault correction to use the FO check as an oracle, and Delvaux [Del22] and Wang *et al.* [WCCL25] attack masked implementations. Correspondingly, several countermeasure families have been proposed: detection with double execution [PP21], shuffling [HPP21], masking, and, specific to the comparison, the *default-fail* `cmov` ordering of Xagawa *et al.* [XIU⁺21].

Table 1 positions our work against prior fault attacks on CCA-secure lattice KEMs along three axes: where the fault is injected, its effect, and the countermeasure it defeats. The distinguishing axis is the last column: ours is the only entry that breaks the *default-fail* `cmov` design, because it manipulates the comparison *value* (forcing *fail* = 0 at the Booleanisation) rather than skipping the conditional move or the equality test. We scope the claim precisely: **Default Correct** defeats the default-fail `cmov` ordering and any defence that protects only a single *fail* flag or the conditional move; with a single fault it does *not* defeat detection/diffusion defences (double execution, redundant comparison, infective computation), which would require correlated multi-fault injection.

Contributions. Taking this pipeline view of the FO comparison, we make the following contributions.

- **A new fault surface in the FO comparison.** We show that the constant-time comparison is a *pipeline* (Booleanisation $\rightarrow$ flag $\rightarrow$ `cmov`) and identify the two’s-complement Booleanisation (`negs/sbc.w` in PQClean and *pqm4* and `rsbs/sbc.w` in reference implementation) as an independently faultable stage upstream of the conditional move. This is a distinct exposure from the `cmov/equality-test` surface targeted by prior fault attacks.
- **Survival against the published countermeasure.** A single sub-instruction glitch on this stage forces *fail* = 0, making decapsulation accept any ciphertext. Because the stage *feeds* the default-fail `cmov` rather than being protected by it, the countermeasure of Xagawa *et al.*, the one defence targeting this comparison, leaves the surface fully open on every hardened configuration we evaluate.

Table 1: Comparison of fault attacks on CCA-secure lattice KEMs. Columns give the fault location, the fault effect, and the countermeasure defeated; the final column is the distinguishing axis of this work.

Work (venue)	Fault location	Fault effect	Countermeasure defeated/Remarks
Ravi <i>et al.</i> [RBRC20] (ePrint’20)	message decoder (EM)	bit flip in m'	unprotected
Pessl-Prokop [PP21] (TCHES’21)	decoding (skip)	ineffective fault	detection, double-exec
Xagawa <i>et al.</i> [XIU ⁺ 21] (ASIACRYPT’21)	equality test / <code>b1 cmov</code>	accept invalid ct	introduces <i>default-fail cmov</i>
Hermelink <i>et al.</i> [HPP21] (INDOCRYPT’21)	ciphertext (FO check)	bit-flip + correction	shuffling
Delvaux “Roulette” [Del22] (TCHES’22)	masked Kyber	multiple	masking
Wang <i>et al.</i> [WCCL25] (SAC’25)	(masked) message decoder	single-bit	masking
This work	comparison Booleanisation (<code>negs/sbc.w</code>)	force <i>fail</i> = 0	<i>default-fail cmov</i>

- **A systemic, cross-ecosystem single point of failure.** We reproduce the fault across 36 hardened configurations (reference, PQClean, and *pgm4*; Kyber-512/768/1024; -00 to -03), showing that the Booleanisation is common to all mainstream builds rather than an artefact of one variant.
- **Key recovery.** We drive the accept-everything behaviour with the standard plaintext-checking oracle [RRCB20, QCZ⁺21] with an η_1 -correct query analysis and a head-to-head comparison against prior fault attacks, and explain why single-fault detection- and diffusion-based defences are *not* bypassed.
- **A producer-side countermeasure.** We replace the two’s-complement Booleanisation with an integrity-checked equivalent that computes the same flag while folding an execution counter into the very instructions that accumulate the mismatch. Any partial instruction or loop skip breaks the counter and yields a non-zero, rejecting flag; only skipping the entire routine bypasses it. We verify this exhaustively over all skip patterns of the routine and measure negligible overhead (well under 0.01% of decapsulation cycles) on all three parameter sets (Section 7).

Experimental setup. We realise the attack on an STM32F303 (ARM Cortex-M4), the NIST-recommended benchmarking target, driven by a low-cost CW1173 ChipWhisperer[®]-Lite over a single USB cable. The fault primitive is a single clock glitch that skips one sub-instruction of the Booleanisation. No chip decapsulation, laser, or electromagnetic probing is required, which keeps the setup simple and repeatable. Chaining the resulting accept-everything oracle, we recover the full secret key of Kyber-512, -768, and -1024, each in a single measured on-device run.

Organization. The remainder of the paper is organised as follows. Section 2 presents preliminaries: notation, the Fujisaki–Okamoto transform, and IND-CCA-secure Kyber-KEM. Section 3 builds the fault-assisted plaintext-checking oracle and the key-recovery procedure. Section 4 develops the **Default Correct** attack on the comparison Booleanisation. Section 5 details the experimental setup, Section 6 reports results across all 36 configurations, and

Section 7 analyses countermeasures and presents our integrity-checked Booleanisation. Section 8 concludes, and Appendix A gives the oracle threshold construction.

2 Preliminaries

2.1 Notations

We use standard notation. $\{0, 1\}^n$ is the set of n -bit strings, $x \leftarrow \{0, 1\}^n$ denotes uniform sampling, $\|$ is concatenation, $\neg$ is logical negation, and $[P]$ is the Iverson bracket (1 if the predicate P holds, 0 otherwise); $\lfloor \cdot \rfloor$ rounds to the nearest integer (ties up). For the KEM we write pk, sk, ct, K for the public key, secret key, ciphertext and session key, m/m' for the encapsulated and decapsulation-side message, r for the encryption randomness, $\tilde{K}$ for the pre-key and z for the implicit-rejection secret, while $\mathcal{G}, \mathcal{H}$ are the hash/expansion functions, J the implicit-rejection PRF, ct_R the re-encrypted reference ciphertext, and *fail* (the Booleanisation of the accumulator $\mathbf{rc}$) the mismatch flag; primed symbols denote decapsulation-side recomputation.

All polynomial arithmetic is carried out in $R_q = \mathbb{Z}_q[X]/(X^n + 1)$ with $n = 256$, $q = 3329$ and module rank $k \in \{2, 3, 4\}$ for Kyber512/768/1024. The secret is the module vector $\mathbf{s} = (s^{(0)}, \dots, s^{(k-1)}) \in R_q^k$: $s^{(p)}$ is its p -th polynomial and $s_i^{(p)}$ the i -th coefficient of that polynomial (parenthesised superscript = polynomial index, subscript = coefficient index, as in Eq. (4)), with $s_i^{(p)} \in [-\eta_1, \eta_1]$ drawn from a centered binomial of parameter η_1 (3 for Kyber512, 2 otherwise). Likewise $\mathbf{u} \in R_q^k$ and $v \in R_q$ are the ciphertext parts, with compression widths d_u, d_v ; when a single coefficient of a fixed target polynomial is isolated (Section 3 onward) we drop the superscript and write s_i . Here $\otimes$ is the negacyclic convolution in R_q and $\mathbf{s}^\top \mathbf{u} = \sum_{p=0}^{k-1} s^{(p)} \otimes u^{(p)}$; we use $\text{Compress}_q(x, d) = \lfloor (2^d/q)x \rfloor \bmod 2^d$, $\text{Decompress}_q(y, d) = \lfloor (q/2^d)y \rfloor \bmod q$, and the message-bit decode $\text{Decode}(w) = \text{Compress}_q(w, 1)$, i.e. $\text{Decode}(w) = 1 \iff w \bmod q \in [q/4, 3q/4]$. The remaining, attack-specific symbols are collected in Table 2.

Table 2: Attack- and oracle-specific notation.

Symbol	Meaning
U, V	attacker-chosen <i>compressed</i> scalars: amplitude $U \in \{0, \dots, 2^{d_u} - 1\}$ on the u -part, offset $V \in \{0, \dots, 2^{d_v} - 1\}$ on the v -part
$\tilde{u}, \tilde{v}$	their <i>decompressed</i> ring values $\text{Decompress}_q(U, d_u), \text{Decompress}_q(V, d_v)$ (used in all R_q arithmetic)
$w[t]$	t -th coefficient of $v - \mathbf{s}^\top \mathbf{u}$ for the crafted ciphertext
$\mathbf{e}_0$	unit vector: 1 at position X^0 , 0 elsewhere
$\mathcal{O}(V_c, U_c)$	plaintext-checking oracle realising the comparator $[s_i \geq c]$
(w, o, e)	glitch width, offset, and ext_offset (Section 5)

All algorithms are assumed to be probabilistic polynomial-time unless stated otherwise.

Kyber is built from a CPA-secure Public Key Encryption (PKE) scheme $\Pi = (\text{KeyGen}, \text{Encrypt}, \text{Decrypt})$ based on the Module-LWE problem and is transformed into an IND-CCA-secure Key Encapsulation Mechanism (KEM) using the Fujisaki–Okamoto (FO) transformation, which combines hashing, message re-encryption, and implicit rejection to resist adaptive chosen-ciphertext attacks. We use the standard syntax and security notions throughout: a PKE is IND-CPA secure if no probabilistic polynomial-time adversary distinguishes encryptions of two chosen plaintexts with non-negligible advantage, and a KEM is IND-CCA secure if no such adversary distinguishes a valid encapsulated key from a random key even given a decapsulation oracle [CS03, HHK17]; the concrete parameters are those of ML-KEM as standardised in FIPS 203 [Nat24]. We recall below only the

components our fault actually targets: the KEM syntax, the FO transform with implicit rejection, and the hash and implicit-rejection machinery.

Definition 1 (KEM). A KEM $\mathcal{K} = (\text{KEYGEN}, \text{ENCAPS}, \text{DECAPS})$ consists of three algorithms:

1. $\text{KEYGEN}() \rightarrow (pk, sk)$ generates a public and secret key pair.
2. $\text{ENCAPS}(pk) \rightarrow (ct, K)$ outputs a ciphertext ct and shared session key K .
3. $\text{DECAPS}(sk, ct) \rightarrow K$ recovers the shared key K from ct using the secret key sk .

Correctness requires that decapsulation of a valid ciphertext recovers the same shared key produced during encapsulation, except with negligible probability.

Definition 2 (FO Transformation [FO99, FO13, HHK17]). The FO transform is a generic method for converting an IND-CPA secure public key encryption scheme into an IND-CCA secure key encapsulation mechanism.

The transformation derives encryption randomness from hashed message material and validates ciphertext integrity through deterministic re-encryption during decapsulation. If ciphertext verification fails, the decapsulation algorithm returns an implicit-rejection value derived from the secret fallback z and the ciphertext. This implicit rejection mechanism prevents adversaries from obtaining useful information through malformed ciphertexts.

Now, we define the hash functions and the implicit-rejection PRF.

Definition 3 (Hash functions and implicit-rejection PRF). Let $\mathcal{H} : \{0, 1\}^* \rightarrow \{0, 1\}^{256}$ be a cryptographic hash function used to derive fixed-length outputs from arbitrary inputs, and let $J : \{0, 1\}^* \rightarrow \{0, 1\}^{256}$ be a pseudorandom function, realised by `rkprf` in the implementation, used for implicit rejection: on a failed ciphertext comparison, decapsulation returns $J(z \parallel ct)$, binding the rejection output to the secret fallback z and the ciphertext. As standardised in ML-KEM (FIPS 203), the shared key on a successful decapsulation is the pre-key $\bar{K}$ itself, so no separate key-derivation step is applied.

In Kyber, hashing is used to remove structure from randomly sampled values and to bind encapsulated keys to ciphertexts.

Let

$$\mathcal{G} : \{0, 1\}^* \rightarrow (\{0, 1\}^{256}, \{0, 1\}^{256})$$

be a hash-based expansion function that deterministically derives the pre-key $\bar{K}$ and encryption randomness r from the input

$$m \parallel \mathcal{H}(pk).$$

2.2 IND-CCA Secure Kyber KEM

Kyber achieves chosen-ciphertext security by applying a Fujisaki-Okamoto style transformation to an underlying CPA-secure encryption scheme [FO99, FO13]. The construction consists of three procedures: key generation, encapsulation, and decapsulation. Refer to Algorithm 1 where we briefly explain each step and how they are connected.

Key Generation. The procedure `CCA.KEYGEN` first samples a random value z . It then runs the underlying CPA-secure key generation algorithm to obtain a public key pk and a secret key sk' . The final secret key is extended by appending the hash of the public key and the random value z . This additional information is later used during decapsulation to handle invalid ciphertexts.

Encapsulation. The procedure CCA.ENCAPS generates a random message m and hashes it to remove any structure. It then derives a pre-key $\bar{K}$ and randomness r using a hash-based function $\mathcal{G}$. The message m is encrypted using the CPA-secure encryption algorithm with randomness r to produce the ciphertext ct . Following ML-KEM, the shared key K is the pre-key $\bar{K}$ itself, and the sender outputs the pair (ct, K) .

Decapsulation. The procedure CCA.DECAPS takes the secret key and a ciphertext as input. It first extracts the stored values including pk , its hash, and z . The ciphertext is decrypted to obtain a message m' . Using this message, the algorithm recomputes the pre-key $\bar{K}'$ and the randomness r' . It then re-encrypts m' to obtain a reference ciphertext ct_R .

The crucial step is the comparison between ct_R and the input ciphertext ct . If they match, the decapsulation is valid and the shared key is the recomputed pre-key $\bar{K}'$. If they do not match, the algorithm instead returns the implicit-rejection key $J(z \parallel ct)$. No further key-derivation step is applied to either outcome.

Algorithm 1: CCA-SECURE KYBER KEM

```

1 procedure  $\text{CCA.KeyGen}$ 
2    $z \leftarrow \{0, 1\}^{256}$ ;
3    $(pk, sk') \leftarrow \text{CPA.KeyGen}()$ ;
4    $sk = (sk' \parallel \mathcal{H}(pk) \parallel z)$ ;
5   return  $(pk, sk)$ ;
6 end

7 procedure  $\text{CCA.Encaps}(pk)$ 
8    $m \leftarrow \{0, 1\}^{256}$ ;
9    $(\bar{K}, r) = \mathcal{G}(m \parallel \mathcal{H}(pk))$ ; /* Generation of pre-key  $\bar{K}$  */
10   $ct = \text{CPA.Encrypt}(pk, m, r)$ ; /* Encryption of message  $m$  using public
    key  $pk$  */
11   $K = \bar{K}$ ; /* Shared key is the pre-key (ML-KEM) */
12  return  $(ct, K)$ ;
13 end

14 procedure  $\text{CCA.Decaps}(sk, ct)$ 
15   $(pk, \mathcal{H}(pk), z) \leftarrow \text{UnpackSK}(sk)$ ;
16   $m' = \text{CPA.Decrypt}(sk, ct)$ ; /* Decryption of ciphertext into message
    */
17   $(\bar{K}', r') = \mathcal{G}(m' \parallel \mathcal{H}(pk))$ ; /* Generation of pre-key  $\bar{K}'$  */
18   $K = \bar{K}'$ ; /* Shared key is the pre-key (ML-KEM) */
19   $ct_R = \text{CPA.Encrypt}(pk, m', r')$ ; /* Re-Encryption of decrypted message
    */
20  if  $(\text{CompareCT}(ct_R, ct) == 0)$  then /* Ciphertext comparison fails */
21     $K = J(z \parallel ct)$ ; /* Implicit rejection */
22  end
23  return  $K$ ;
24 end

```

3 Fault-Assisted Binary Oracle

The fault-assisted plaintext-checking (PC) / key-mismatch oracle for lattice KEMs is well established in the literature [RRCB20, QCZ⁺21]; once the Default Correct fault has turned decapsulation into an accept-everything primitive, we invoke it in exactly this standard

form. We recall the construction here for completeness and to keep the presentation self-contained, fixing its parameter dependence (Section 3.2) and reporting the per-parameter query counts.

Kyber decapsulation internally decodes the message as $m' = \text{Decode}(v - \mathbf{s}^\top \mathbf{u})$ as shown in Eq. (4), where $\mathbf{s}^\top \mathbf{u} = \sum_p s^{(p)} \circledast u^{(p)}$ is the negacyclic convolution in $R_q = \mathbb{Z}_q[X]/(X^{256} + 1)$. Coefficient-wise, for Kyber768 ($k = 3$),

$$m'[i] = \text{Decode}\left(v[i] - (s^{(0)} \circledast u^{(0)} + s^{(1)} \circledast u^{(1)} + s^{(2)} \circledast u^{(2)})[i]\right). \quad (1)$$

To isolate a single secret coefficient $s_i^{(p)}$ at a time, we craft a sparse ciphertext without any side-channel information on m : the p -th u -polynomial carries a single non-zero *compressed* coefficient $U \in \{0, \dots, 2^{d_u} - 1\}$ at position $(256 - i) \bmod 256$, i.e. position 0 when $i = 0$ (all other u -polynomials zero), and the v -part carries a single *compressed* constant $V \in \{0, \dots, 2^{d_v} - 1\}$. Decapsulation decompresses these to $\tilde{u} = \text{Decompress}_q(U, d_u)$ and $\tilde{v} = \text{Decompress}_q(V, d_v)$ before the ring subtraction, so in R_q it decodes

$$m' = \text{Decode}\left(\tilde{v} X^0 - \left(s^{(p)} \circledast \tilde{u} X^{256-i}\right) \bmod (X^{256} + 1)\right) = \mathbf{e}_0, \quad (2)$$

where $\mathbf{e}_0$ denotes the unit vector that is 1 at position X^0 and 0 elsewhere. Both sides are decoded binary vectors, so the equation is type-consistent; whether the X^0 bit is set is exactly the oracle output, which depends on $s_i^{(p)}$ and is made precise in Section 3.2.

$$\begin{aligned} s^{(p)} \circledast \tilde{u} X^{256-i} &= \tilde{u} \sum_{j=0}^{255} s_j^{(p)} X^{j+256-i} \\ &= \tilde{u} s_0^{(p)} X^{256-i} + \tilde{u} s_1^{(p)} X^{257-i} + \dots + \tilde{u} s_{255}^{(p)} X^{511-i} \pmod{X^{256} + 1} \end{aligned} \quad (3)$$

Reducing modulo $X^{256} + 1$ (so that $X^{256} \equiv -1$), every term of Eq. (3) except the one whose exponent is $\equiv 0$ lands on a position $t \neq 0$; the surviving constant-coefficient contribution is the $j = i$ term, and its sign is fixed by whether that term wraps. For $i \in \{1, \dots, 255\}$ the exponent is 256, which reduces to $X^{256} \equiv -1$, so the contribution is $-\tilde{u} s_i^{(p)}$. For $i = 0$ the compressed coefficient physically occupies position $256 \bmod 256 = 0$, so the u -part is the monomial $\tilde{u} X^0$ (Eq. (5)); its $j = 0$ term has exponent 0, does not wrap, and contributes $+\tilde{u} s_0^{(p)}$. (Encoding the formal $X^{256} \equiv -1$ instead, by placing $q - \tilde{u}$ at position 0, would restore a uniform minus sign; we keep $+\tilde{u}$ at X^0 and mirror the $i = 0$ thresholds in Appendix A.) Thus setting the u -part to $\tilde{u} X^{256-i}$ (compressed value U) isolates the single secret coefficient $s_i^{(p)}$ into the X^0 position, while the offset $\tilde{v}$ on v (compressed value V) sets the decode threshold (Eqs. (5) and (6) give the $i = 0$ and $i = 1$ instances). Choosing (U, V) as in Section 3.2 keeps every other decoded coefficient at 0, so the flip of $m'[0]$ is a clean binary distinguisher for $s_i^{(p)}$, for any coefficient at any position. The corresponding plaintext, in a white-box CCA setting, is obtained from the Kyber-PKE known-answer tests (KATs), with $U \in \{0, \dots, 2^{d_u} - 1\}$ and $V \in \{0, \dots, 2^{d_v} - 1\}$.

$$\text{For } i \in \{1, \dots, 255\} : \quad \begin{pmatrix} m'_0 \\ m'_1 \\ \vdots \\ m'_{255} \end{pmatrix} = \text{Decode} \left(\begin{pmatrix} v_0 \\ v_1 \\ \vdots \\ v_{255} \end{pmatrix} - \sum_{p=0}^{k-1} \begin{pmatrix} s_0^{(p)} \\ s_1^{(p)} \\ \vdots \\ s_{255}^{(p)} \end{pmatrix} \circledast \begin{pmatrix} u_0^{(p)} \\ u_1^{(p)} \\ \vdots \\ u_{255}^{(p)} \end{pmatrix} \right) \quad (4)$$

$$\text{For } i = 0 : \begin{pmatrix} m'_0 \\ m'_1 \\ \vdots \\ m'_{255} \end{pmatrix} = \text{Decode} \left(\begin{pmatrix} \tilde{v} \\ v_1 \\ \vdots \\ v_{255} \end{pmatrix} - \begin{pmatrix} s_0^{(p)} \\ s_1^{(p)} \\ \vdots \\ s_{255}^{(p)} \end{pmatrix} \circledast \begin{pmatrix} \tilde{u} \\ u_1^{(p)} \\ \vdots \\ u_{255}^{(p)} \end{pmatrix} \right) \quad (5)$$

$$\text{For } i = 1 : \begin{pmatrix} m'_0 \\ m'_1 \\ \vdots \\ m'_{255} \end{pmatrix} = \text{Decode} \left(\begin{pmatrix} \tilde{v} \\ v_1 \\ \vdots \\ v_{255} \end{pmatrix} - \begin{pmatrix} s_0^{(p)} \\ s_1^{(p)} \\ \vdots \\ s_{255}^{(p)} \end{pmatrix} \circledast \begin{pmatrix} u_0^{(p)} \\ \vdots \\ u_{254}^{(p)} \\ \tilde{u} \end{pmatrix} \right) \quad (6)$$

3.1 Key Recovery

Now for the key recovery, the attack follows a decision tree to feed the crafted ciphertext to Kyber decapsulation in order of CBD probability of candidate. The decision tree is η_1 -dependent: it resolves each secret coefficient within at most 3 queries for Kyber768/1024 ($\eta_1 = 2$, $s_i \in [-2, +2]$) and within at most 4 queries for Kyber512 ($\eta_1 = 3$, $s_i \in [-3, +3]$).

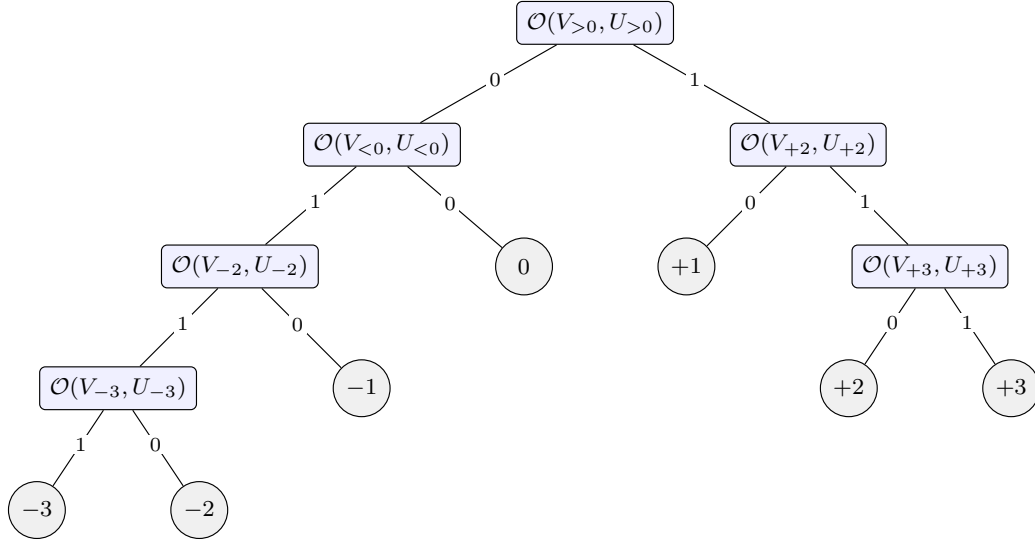


Figure 1: Key-recovery decision tree for one secret coefficient s_i , shown for Kyber512 ($\eta_1 = 3$, $s_i \in [-3, +3]$). Each internal node is one oracle query $\mathcal{O}(V_c, U_c)$; its returned bit selects the outgoing edge (1 or 0), and a leaf gives the recovered coefficient. The positive tests read $\mathcal{O} = [s_i \geq c]$ and the negative tests their complement (Appendix A). Queries are issued in decreasing centered-binomial (CBD) probability so that the likeliest values are resolved first, in at most 4 queries. For Kyber768/1024 ($\eta_1 = 2$, $s_i \in [-2, +2]$) the two $\mathcal{O}(V_{\pm 3}, U_{\pm 3})$ nodes and the $s_i = \pm 3$ leaves are pruned, so the $\mathcal{O}(V_{\pm 2}, U_{\pm 2})$ “1” branches terminate directly at $s_i = \pm 2$ in at most 3 queries. All branch outcomes are verified exhaustively in Appendix A.

The recovery machinery that consumes this bypass is the well-known plaintext-checking (PC) / key-mismatch oracle for lattice KEMs [RRCB20, QCZ⁺21]. We recall it here for completeness, in the exact form our fault instantiates, since the crafted ciphertexts and the distinguisher depend on the parameters of Section 3.2.

Offline characterisation and precomputation. The attack learns about the decrypted message not by observing the Kyber-PKE plaintext directly, but through an offline characterisation of the decryption function over all compressed ciphertext values $U \in \{0, \dots, 2^{d_u} - 1\}$ and $V \in \{0, \dots, 2^{d_v} - 1\}$. This identifies the (U, V) pairs for which the target message bit m'_0 flips as a function of the underlying secret coefficient (Appendix A). Because $\mathcal{G}$ and $\mathcal{H}$ are public, the attacker precomputes the two session keys the device can return on a successful bypass: the baseline key K_{base} for the all-zero message ($m'_0 = 0$) and the triggered key K' for the flipped bit ($m'_0 = 1$). In the attacked build (Fig. 4) this returned key is the first `KYBER_SYMBYTES` of $\mathcal{G}(m' \parallel \mathcal{H}(pk))$; no honest encapsulated message is involved.

Online distinguisher. The adversary injects the `Default Correct` fault to bypass verification, submits each crafted ciphertext to the decapsulation oracle, and compares the returned key against the two precomputed references. Each attempt has two outcomes. If the glitch fails, verification rejects and decapsulation returns the implicit-rejection key $J(z \parallel ct)$ (line 9 of Fig. 4); being a secret-dependent value it matches neither reference, so the attacker reads any output outside $\{K_{\text{base}}, K'\}$ as a failed injection and retries. If the glitch succeeds, the output equals K_{base} or K' , yielding a clean binary distinguisher for m'_0 , the flip of Eq. (2), without revealing the message itself.

Coefficient recovery. Each crafted ciphertext is built (Appendix A) so that the m'_0 -flip occurs precisely when the secret coefficient meets a chosen threshold $[s_i \geq c]$; a successful query therefore evaluates one comparator against s_i . Ordering these comparators into the decision tree of Fig. 1 over the range $[-\eta_1, \eta_1]$ resolves s_i exactly, and issuing the queries in decreasing centered-binomial (CBD) probability resolves the likeliest values first: at most 4 logical queries for Kyber512 ($\eta_1 = 3$, seven candidate values) and 3 for Kyber768/1024 ($\eta_1 = 2$, five values). Weighting each leaf's depth in Fig. 1 by its CBD probability gives the expected per-coefficient cost explicitly. For Kyber512 ($\eta_1 = 3$) the CBD counts over $s_i = -3, \dots, +3$ are $(1, 6, 15, 20, 15, 6, 1)/64$ and the corresponding leaf depths are $(4, 4, 3, 2, 2, 3, 3)$, so

$$\mathbb{E}[L] = \frac{4 \cdot 1 + 4 \cdot 6 + 3 \cdot 15 + 2 \cdot 20 + 2 \cdot 15 + 3 \cdot 6 + 3 \cdot 1}{64} = \frac{164}{64} = 2.5625.$$

For Kyber768/1024 ($\eta_1 = 2$) the counts over $s_i = -2, \dots, +2$ are $(1, 4, 6, 4, 1)/16$ and, once the ± 3 nodes are pruned, the depths are $(3, 3, 2, 2, 2)$, so

$$\mathbb{E}[L] = \frac{3 \cdot 1 + 3 \cdot 4 + 2 \cdot 6 + 2 \cdot 4 + 2 \cdot 1}{16} = \frac{37}{16} = 2.3125.$$

Thus the expected cost is ≈ 2.56 queries per coefficient for Kyber512 and ≈ 2.31 for Kyber768/1024; over the $256k$ secret coefficients this totals ≈ 1312 , ≈ 1776 , and ≈ 2368 logical queries for Kyber512 (Level 1), Kyber768 (Level 3), and Kyber1024 (Level 5), i.e. $k = 2, 3, 4$ respectively.

Cost. The logical query count bounds the information-theoretic work, but each logical query may require hundreds to thousands of physical fault-injection attempts before a verification bypass succeeds, owing to the low success probability of the sub-instruction skip. Each crafted ciphertext is therefore re-injected and glitched until the FO check is bypassed; the end-to-end cost of a full key recovery is dominated by this physical glitching effort rather than the logical query count, and is quantified as `Num_executions` in Table 3.

Measured end-to-end recovery. Table 3 reports full key-recovery runs on the ChipWhisperer-Lite setup of Section 5, each recovering the entire secret vector sk in a single uninterrupted

run. Kyber1024 completed in 3 h 16 m 06 s ($\approx 11,766$ s) using 6,082 physical fault attempts, of which 2,382 produced the intended single-instruction skip; Kyber768 completed in 13 h 21 m 24 s ($\approx 48,084$ s) using 13,978 attempts with 1,784 successful skips; Kyber512 completed in 21 h 55 m 11 s ($\approx 78,911$ s) using 24,195 attempts with 1,320 successful skips. Crucially, the number of successful skips matches the logical-query counts predicted above (≈ 2368 for Kyber1024, ≈ 1776 for Kyber768, ≈ 1312 for Kyber512) to within 1%, confirming that each successful bypass answers exactly one oracle query and that the η_1 -dependent decision tree closes as analysed. For Kyber1024 the expected execution budget of 6,083 likewise matches the 6,082 measured attempts.

The variation is an artefact of glitch-parameter tuning: our objective was to obtain a reliable *first* instruction skip with parameters, namely, width, offset, ext_offset (w, o, e) combination per build, not to maximise its success probability. The rate at each security level can be raised by an exhaustive sweep for the optimal (w, o, e) combination because such tuning is an engineering optimisation rather than a cryptographic contribution. It is tedious to carry out per parameter set, we did not pursue it and report the key recovery at the first working parameter set.

Table 3: Full key recovery and fault characteristics for each Kyber security level.

Parameter set	Fault complexity (instructions)	Skips	Resets/ crashes	Normal	Num_exec	Wall-clock
Kyber512	1–2	1320	12647	10228	24,195	21 h 55 m 11 s
Kyber768	1–2	1784	7117	5077	13,978	13 h 21 m 24 s
Kyber1024	1–2	2382	863	2837	6,082	3 h 16 m 06 s

3.2 Oracle instantiation

Following the standard PC/key-mismatch ciphertext crafting [RRCB20, QCZ⁺21], the attacker sends the sparse ciphertext of Eq. (2): a single compressed amplitude U on the p -th u -polynomial at position $256 - i$ and a single compressed offset V on v . Every decoded message bit then vanishes except at position 0, where the surviving bit is $\text{Decode}(\tilde{v} + \tilde{u} s_i)$. This is a single step that increases monotonically with the target coefficient s_i . By placing the decompressed offset $\tilde{v}$ just below the $q/4$ decode boundary, the device answers the threshold query $\mathcal{O}(V_c, U_c) = [s_i \geq c]$ (“is $s_i \geq c$?”). The isolation and feasibility conditions that make this exact for every parameter set are derived and machine-verified in Appendix A.

The oracle response is a monotone staircase in the target coefficient: the query $\mathcal{O}(V_c, U_c)$ returns 1 exactly when $s_i \geq c$. This staircase is resolved by the η_1 -dependent decision tree of Fig. 1 in at most 4 ($\eta_1 = 3$) or 3 ($\eta_1 = 2$) queries per coefficient, giving the per-parameter query counts of Section 3. The concrete per-level (U, V) instantiation and its exhaustive verification are given in Appendix A (Table 5).

4 The Default Correct Attack on Kyber

Kyber’s chosen-ciphertext security rests on a single accept/reject decision: the FO comparison collapses the entire ciphertext check to one Booleanised flag *fail*, and the conditional move (`cmov`) releases the session key unless *fail* = 1 marks a mismatch (Fig. 4). Our attack targets this flag directly: if *fail* can be forced to 0, decapsulation succeeds for *any* crafted ciphertext, whether or not it matches. Crucially, the flag is *produced* upstream by the two’s-complement Booleanisation and only *consumed* by the `cmov`, so this is a fault surface independent of the conditional move itself. As a consequence, any countermeasure

that protects only the conditional move or a single *fail* flag is bypassed, including the default-fail `cmov` ordering of Xagawa *et al.* [XIU⁺21], which pre-loads the session key with a pseudo-random z so that a skipped `cmov` leaks nothing: it hardens the *consumer* of a value we corrupt before it is ever consumed. This bypass is scoped, however: a single forced *fail* = 0 does *not* defeat detection or diffusion defences, which require correlated multi-fault injection and to which we return in Section 7.

4.1 *rc* assignment for ciphertext comparison

In the fused build, the comparison result is accumulated into a single variable `rc`, declared as a `uint64_t` and initialised to 0 in `indcpa_enc_cmp` (Fig. 5); its Booleanised value becomes the mismatch flag *fail* consumed by `crypto_kem_dec` (Fig. 4). During re-encryption, the recomputed polynomials are serialised and compared byte-by-byte against the input ciphertext through `cmp_polyvec_compress` (the *u*-part) and `cmp_poly_compress` (the *v*-part), each contributing any mismatch into the same `rc` via a constant-time bitwise accumulation. The two’s-complement idiom then Booleanises `rc`: `rc = ~rc + 1` propagates any set bit towards the most significant bit (MSB), and the shift `rc >>= 63` brings the MSB to the least significant bit (LSB), so the returned `(unsigned char)rc` is 1 on mismatch and 0 on match. The reference and PQClean builds compute the same flag with a separate `verify()` routine that loops over the whole ciphertext (Fig. 6) and uses the identical Booleanisation `(~(uint64_t)r) >> 63`; in every case the entire accept/reject decision is reduced to one byte. This branch-free arithmetic reduction is chosen for efficiency and constant-time behaviour: any equivalent formulation using conditional branching would require more instructions and would leak timing information [SS93].

In case of a match as shown in Figure 2a, the two’s complement operation doesn’t affect the register. The MSB remains 0 since the addition of 1 to the `~rc` in the two’s complement system discards the carry bit. So, the `rc` becomes 0 after the shift. During a mismatch case in Figure 2b, the MSB of the `rc` gets flipped from 0 to 1. Therefore, the `rc` becomes 1 after the shift, leading to decapsulation failure.

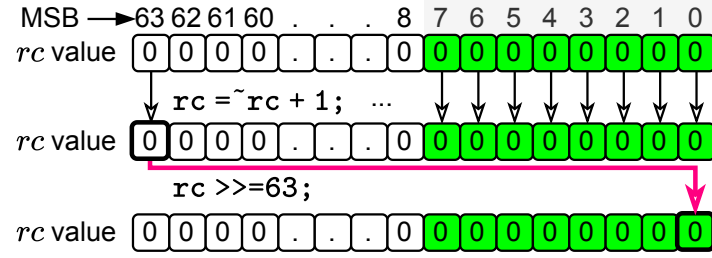
Concretely, we target `indcpa.c::indcpa_enc_cmp` (fused build) and `verify.c::verify` (reference / PQClean) of *pqm4*; the Booleanisation `rc = ~rc + 1; rc >>= 63` compiles to the `negs/sbc.w/lsrcs` sequence of Fig. 7. The same reduction appears, with per-build register allocation, in the reference implementation (Fig. 8; the negation is emitted as `rsbs`) and in the PQClean `verify()` routine (Fig. 9), confirming that the target sequence is common to all three code bases.

Remark 1. The exposure is not specific to Kyber. The FrodoKEM reference implementation [The21] realises its FO comparison with the same three stages: `ct_verify` ORs the word-wise differences into an accumulator and reduces it with the two’s-complement idiom `(~(int16_t)r) >> 15` to a 0/−1 flag, and a constant-time `ct_select` consumes that flag. We have not evaluated FrodoKEM experimentally.

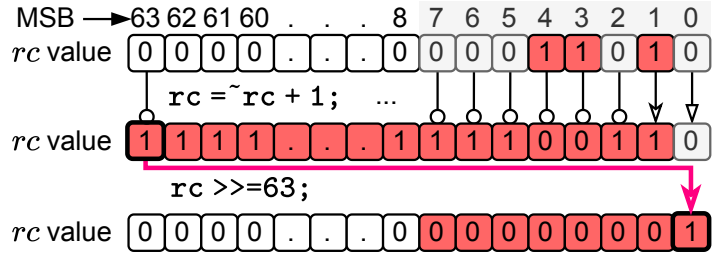
4.2 The Proposed Attack

As illustrated in Figure 5, the Default Correct attack exploits the verification stage of Kyber decapsulation. Disrupting the two’s complement sequence by skipping single or multiple instructions can bypass the path to decryption failure. Whether the sequence is entirely skipped or heavily corrupted, as long as the least significant byte evaluates to 0, the variable *fail* will store 0. This leads to a false decapsulation success, causing the device to reveal the session key K' regardless of the ciphertext’s validity. Refer to Figure 7 for the assembly code. The constant-time Booleanisation (`rc = ~rc + 1;`) and shift (`rc >>= 63`) are executed via a strict three-instruction sequence on the ARM Cortex-M4.

First, the `negs` instruction performs a two’s complement negation on the lower half of the state, generating a borrow flag if any byte-wise mismatch is present.



(a) Successful decapsulation in normal execution.



(b) Decapsulation failure in normal execution.

Figure 2: Normal (unfaulted) decapsulation: the `rc` Booleanisation on a matching ciphertext (success) and a mismatching ciphertext (failure).

Next, the `sbc.w` (Subtract with Carry) instruction consumes this borrow to mathematically flip the upper state to all ones, ensuring the mismatch signal reliably propagates to the MSB.

Finally, the `lsrs` (Logical Shift Right) instruction shifts the MSB to LSB where the variable `fail` stores the decision 1 for failure, 0 for success. Although the C source writes `rc  $\gg=$  63` on a 64-bit value, the compiled sequence shifts the *high* 32-bit word by 31 (`lsrs r4, r4, #31`); the two are equivalent because `rc` is held across two 32-bit registers and only its most significant bit determines `fail`.

Why the Booleanisation is a stage independent of the `cmov`. The accept/reject decision is not computed at one instruction but flows through two stages joined by a single data dependency. The first stage i.e., the Booleanisation, lives inside the comparison routine (`indcpa_enc_cmp` in the fused build, Fig. 5; `verify()` in the reference/PQClean build, Fig. 6) and reduces the whole-ciphertext mismatch accumulator `rc` to the one-byte flag `fail` via the `negs/sbc.w/lsrs` sequence of Fig. 7. The second stage i.e., the selection, lives in `crypto_kem_dec` (Fig. 4), which consumes that byte in `cmov(ss, kr, KYBER_SYMBYTES, 1 - fail)` to release either the true key material or the implicit-rejection value. The two stages are separated in the instruction stream, in different functions executed at different times, and the *only* channel between them is the value of `fail`: the selection stage never recomputes, re-derives, or validates the mismatch, but treats `fail` as an opaque input.

Xagawa *et al.*'s default-fail ordering is a transformation of the *selection* stage alone: it pre-loads the output with the rejection value so that a *skipped* `cmov` releases nothing. It leaves the Booleanisation, and the `1 - fail` argument it feeds, untouched. The Default Correct fault therefore does not race, skip, or defeat the hardened `cmov`; it corrupts `fail` one stage earlier, and the hardened `cmov` then performs its constant-time selection *correctly* on the corrupted input, releasing the key because, as far as the selection stage can tell, the comparison reported a match. Independence is thus not merely asserted but visible in the

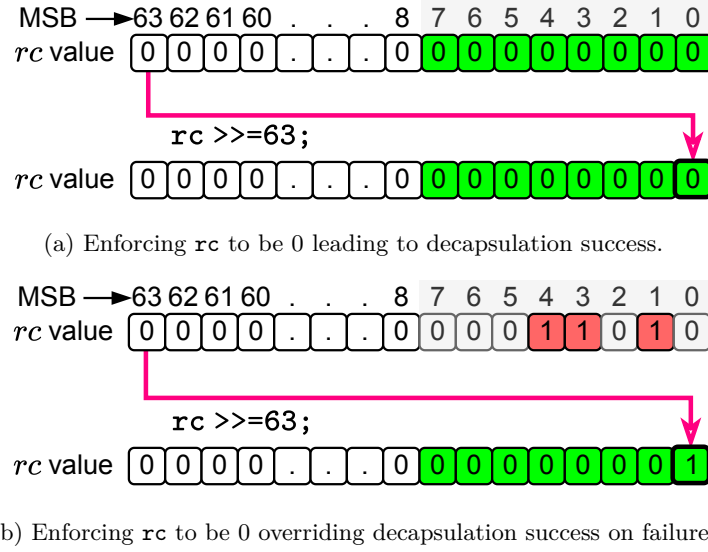


Figure 3: Effect of forcing `rc = 0` on the decapsulation procedure, showing faulty execution.

data flow: a countermeasure confined to the consumer of *fail* cannot constrain the producer of *fail*, and no reordering of the selection stage changes which byte the Booleanisation hands it.

The fault model admits the following scenarios:

- A very intuitive scenario is where a fault skips only the `negs` instruction at line 2, such that there is no borrow generated for any byte-wise mismatch¹. The `sbc.w` instruction at line 3 then flips nothing in the upper word, and the subsequent `lsrs` shift to LSB stores *fail* as 0, leading to decapsulation success.
- The glitch skips both the `negs` and `sbc.w` instructions at line 2 and line 3 as shown in Figure 3, resulting in the `uint64_t rc` being 0. The subsequent `lsrs` shift to LSB then stores *fail* as 0.

```

1  int crypto_kem_dec(uint8_t *ss, const uint8_t *ct, const uint8_t *sk) {
2  int fail;
3  uint8_t buf[2 * KYBER_SYMBYTES], kr[2 * KYBER_SYMBYTES];
4  const uint8_t *pk = sk + KYBER_INDCPA_SECRETKEYBYTES;
5
6  indcpa_dec(buf, ct, sk);
7  hash_g(kr, buf, 2 * KYBER_SYMBYTES);
8  fail = indcpa_enc_cmp(ct, buf, pk, kr + KYBER_SYMBYTES);
9  rkprf(ss, sk + KYBER_SECRETKEYBYTES - KYBER_SYMBYTES, ct);
10 cmov(ss, kr, KYBER_SYMBYTES, (uint8_t) (1 - fail));
11 return 0;
12 }
```

Figure 4: Kyber Decapsulation inside `kem.c`

¹The same applied to the `rsbs` instruction for the reference implementation.

```

1  unsigned char indcpa_enc_cmp(const unsigned char *c,
2                               const unsigned char *m,
3                               const unsigned char *pk,
4                               const unsigned char *coins) {
5      uint64_t rc = 0;
6      /* ... re-encrypt m under (pk, coins) on the fly ... */
7      /* u-part: compare recomputed polyvec against c[0 .. c2) */
8      rc |= cmp_polyvec_compress(c, /* recomputed u */ ...);
9      /* v-part: compare recomputed poly against c[c2 ..) */
10     rc |= cmp_poly_compress(c + KYBER_POLYVECCOMPRESSEDBYTES,
11                             /* recomputed v */ ...);
12     rc = ~rc + 1;      /* Booleanise: negate (negs) + borrow (sbc.w) */
13     rc >>= 63;         /* shift MSB -> LSB (lsrs): 1=mismatch, 0=match */
14     return (unsigned char)rc;
15 }

```

Figure 5: Fused comparison (*pqm4* memory-optimized build): a single `rc` accumulates mismatches over *both* the *u* (polyvec) and *v* (poly) parts of the ciphertext before the two’s-complement Booleanisation. The ... mark re-encryption operands omitted for brevity. (`c2 = KYBER_POLYVECCOMPRESSEDBYTES`.)

```

1  /* reference / PQClean: separate verify() over the whole ct */
2  int verify(const uint8_t *a, const uint8_t *b, size_t len) {
3      uint8_t r = 0;
4      for (size_t i = 0; i < len; i++)
5          r |= a[i] ^ b[i];      /* accumulate any byte mismatch */
6      return (-(uint64_t)r) >> 63; /* Booleanise: 1=mismatch, 0=match */
7  }

```

Figure 6: Reference/PQClean comparison inside `verify.c`: `verify(ct, cmp, KYBER_CIPHERTEXTBYTES)` reduces the whole-ciphertext mismatch to one byte via `-(uint64_t)r >> 63`. The Default Correct fault targets this same Booleanisation; only its exact instruction layout differs per build.

5 Experimental Setup

To execute the given idea, we use CW1173 ChipWhisperer-Lite, an integrated board with both target and glitch hardware [New] as shown in Figure 10. The board connects to the workstation over a single micro-USB cable. The implementation of Kyber is optimized for the Cortex-M4 microcontroller taken from *pqm4* [KRSS19], an open-source library for PQC schemes on the ARM Cortex-M4 microcontroller. As per NIST recommendations, we use the STM32F303, which is equipped with an ARM Cortex-M4 microcontroller, for benchmarking the implementation of Kyber [Del22]. The on-board STM32F303 has no support for true-random number generation (*TRNG*) [Tea22]. Thus, the pseudo-random number generator (*PRNG*) used in our paper is SURF PRNG found in the *pqm4* library. The PC sends crafted ciphertexts, where ChipWhisperer-Lite controls the communication between the Device under Test (DUT) and the PC. We evaluate the attack across compiler optimisation levels -O0--O3 (Table 4), confirming that it is not an artefact of an unoptimised build. The three code bases (reference, PQClean, and *pqm4*, scheme `crypto_kem/kyber{512,768,1024}`) are compiled with `arm-none-eabi-gcc 13.2.1` (release 13.2.rel1) using flags `-mthumb -mcpu=cortex-m4 -mfloat-abi=soft -mfpu=vfp`. We target the STM32F303 because it carries the same ARM Cortex-M4 core as the *pqm4*-standard STM32F407 and is integrated on the CW-Lite; as Table 4 shows, the pulse width and offset stay essentially constant across builds while only the `ext_offset` is re-profiled, indicating that the glitch parameters port across implementations and optimisation levels. The on-board STM32F303 target

```

1  bl    8004c10
2  negs  r4, r4
3  sbc.w r4, r5, r5, lsl #1
4  bl    8004c1e ;
5  lsrs  r4, r4, #31
6  ldr   r3, [pc, #44]
7  mov   r0, r4

```

Figure 7: Assembly code of Kyber’s byte-wise comparison in the *pqm4* (*indcpa.c*)

```

1  rsbs  lr, lr, #0
2  sbc.w r5, r5, r5
3  lsrs  r5, r5, #31
4  movs  r6, #0

```

Figure 8: Assembly code of Kyber’s byte-wise comparison in the *reference* implementation (*indcpa.c*).

runs at 7.37 MHz. However for glitching we utilize its maximum sampling rate of 29.48 MS/s. Trigger is added before and after the sensitive operation to proceed with standard Targeted-Fault injection [RCDB22]. The glitch is created by XORing a short impulse with the master clock on which Kyber runs, and is configured using three parameters. A *width* (w) of the pulse defines how long the glitch lasts on the DUT, an *offset* (o) is the timing of the fault with respect to the rising clock edge of the master clock to that of glitch pulse, and lastly an *ext_offset* (e), which is expressed as the delay applied by the glitch hardware between receiving the trigger and inserting the glitch [Pay20, Del22]. Through a rigorous parameter sweep of (w, o, e) , it results in an instruction skip exactly where we want to. The two parameters (w, o) are responsible for the intensity that must be carefully balanced. If too low, no data is faulted, and if too high, our target crashes. To avoid measurement artifacts, the trigger-toggling instrumentation is kept off the decapsulation return-value path, so that a residual instrumentation value (such as a GPIO base address left in the return register) can never be misread as a successful decapsulation.

Clock configuration and transferability. The STM32F303 target is driven by an external, fixed 7.37 MHz clock supplied by the ChipWhisperer-Lite capture board; the target’s internal RC oscillator and PLL are not used, so the core runs directly on the supplied clock. The glitch is generated in the capture hardware by XORing a short impulse into this clock line (the `clock_xor` glitch output), and the capture board samples the target at 29.48 MS/s, i.e. $4\times$ the target clock. We stress that 29.48 MS/s is the sampling rate, not the target frequency. Because the fault is delivered on the clock line, the primitive presupposes physical control of the target’s clock: it applies as-is to any configuration in which the core is clocked from an externally accessible source, and would have to be re-characterised for a target running from an internal oscillator or through a PLL, where the glitch must either survive the clock-conditioning path or be replaced by a different injection channel (e.g. voltage or electromagnetic). Likewise, different clock frequencies, clock architectures, or MCU families require re-characterising (w, o, e) ; Section 6.1 shows that (w, o) behave as rig-level constants while e is a small per-build offset.

6 Results

To demonstrate that the **Default Correct** fault is not tied to a single build, we reproduce it across three independent code bases, namely the *pqm4* optimized implementation, the

```

1  movs    r5, #0
2  negs    r4, r4
3  sbc.w   r4, r5, r5
4  lsrs    r4, r4, #31

```

Figure 9: Assembly code of Kyber’s byte-wise comparison in the PQClean implementation (`verify.c`).

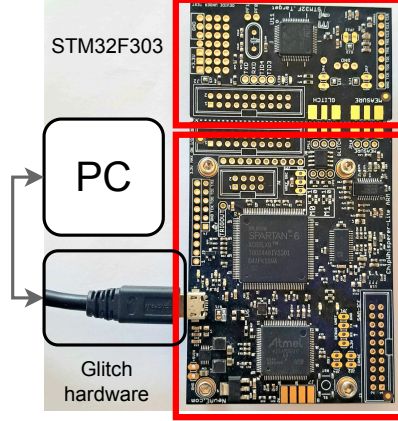


Figure 10: Experimental setup for clock glitching.

NIST reference implementation, and the PQClean implementation, for all three parameter sets (Kyber512, Kyber768, Kyber1024) and at every compiler optimization level (`-O0`, `-O1`, `-O2`, `-O3`), a total of 36 configurations. For each of these configurations, we perform 1000 fault injections on the STM32F303 target. Every configuration is built with the *default-fail* `cmov` ordering. Every attempt targets two’s-complement Booleanisation (fault complexity 1–2 instructions, `repeat = 1`) using the glitch parameters (w, o, e) listed in Table 4. Using these faults, we mount the full key-recovery attack with our **Fault-Assisted Binary Oracle** and recover the long-term secret sk . Table 3 reports complete single-run recoveries for all three parameter sets for Kyber512/768/1024. The per-injection skip-success rate measured over these full runs is found to be 5.5%/12.8%/39.2% respectively. The remaining injections either leave the execution unaffected or crash the target, outputting corrupted values.

The total number of physical executions is denoted by `Num_executions` which is the sum of successful skips, crashes, and normal runs. We report `Num_executions` as 24,195 for Kyber512, 13,978 for Kyber768, and 6,082 for Kyber1024 (Table 3). As expected, `Num_executions` scales inversely with the skip-success rate, so the lowest-rate Kyber512 run required the most executions and wall-clock time (21 h 55 m) despite its smaller secret, while the high-rate Kyber1024 run completed in 3 h 16 m. We summarise the attack characteristics by the tuple (`Glitching`, `Communication_DUT_IO`, `Targeted_Fault`, 1–2, 6,082–24,195).

Table 4 reports, for each configuration, the glitch parameters, namely pulse width (w), offset (o), and `ext_offset` (e), that reproduce the fault; the meaning of (w, o, e) is as defined in Section 5. The width and offset stay essentially constant ($w \approx 1.49$ – 1.51 , $o \approx 0.98$ – 1.00) across all builds, confirming that the fault depends on the timing window of the two’s-complement sequence rather than on the surrounding implementation; only the `ext_offset` (e), which locates the Booleanisation in time relative to the trigger, must be re-profiled per build because the optimizer relocates the instruction sequence.

Table 4: Glitch parameters (pulse width w , offset o , ext_offset e) and the measured per-injection skip-success rate ($s\%$) that reproduce the **Default Correct** fault across the *pqm4*, NIST reference, and PQClean implementations of Kyber at all four optimization levels (STM32F303). The $s\%$ column reports the minimum and maximum number of successful skips observed across ten independent runs of 100 fault injections each (1000 injections per configuration); since each run is 100 injections, the skip count equals the per-injection skip-success rate in percent. The pulse width and offset are essentially constant throughout ($w \approx 1.49\text{--}1.51$, $o \approx 0.98\text{--}1.00$) and are therefore omitted; only the ext_offset e (cycles) and the skip rate $s\%$ are tabulated. The parameters (w, o, e) are defined in Section 5.

Implementation	Scheme	-00		-01		-02		-03	
		e	$s\%$	e	$s\%$	e	$s\%$	e	$s\%$
<i>pqm4</i>	Kyber512	10	16–28	9	1–5	8	11–26	10	5–9
	Kyber768	8	7–13	26	6–14	16	4–15	12	6–19
	Kyber1024	8	5–13	4	8–17	11	10–23	10	12–22
Reference	Kyber512	11	3–11	11	34–53	5	40–51	6	31–44
	Kyber768	10	8–18	1	37–52	6	9–16	11	39–49
	Kyber1024	10	7–17	1	46–58	9	41–50	7	32–46
PQClean	Kyber512	7	1–3	1	45–51	4	38–49	7	41–48
	Kyber768	12	9–15	1	45–55	7	10–19	7	28–46
	Kyber1024	6	7–15	28	22–32	7	6–15	7	10–18

6.1 Interpreting the glitch parametrisation

Table 4 separates the fault into two nearly independent kinds of parameters, and that split is itself informative about the mechanism. The pulse width w and offset o set the glitch *intensity*: how strongly the injected impulse perturbs the clock edge, and hence whether the target instruction is corrupted at all. These are properties of the analog injection path, not of the code, which is why they stay in a narrow band ($w \approx 1.49\text{--}1.51$, $o \approx 0.98\text{--}1.00$) across all 36 configurations and all three code bases. Thus, they do not transfer between ChipWhisperer units. However, a different board will realise the same intensity at slightly different (w, o) .

The ext_offset e sets when the glitch fires, in cycles from the hardware trigger to the injected pulse. In our harness the trigger is raised immediately before the Booleanisation and lowered immediately after it (`trigger_high()/trigger_low()` around `rc = ~rc + 1; rc >>= 63` in the fused *pqm4* comparison, and around `r = ~(uint64_t)r >> 63` in the reference/PQClean `verify()`). Crucially, the byte-wise mismatch accumulation, the part of the comparison whose length grows with the ciphertext, lies *before* the trigger. As a result e measures only the short, fixed distance from the trigger to the two’s-complement sequence and does *not* encode the ciphertext-comparison time. This is exactly what the table shows. At a fixed optimisation level, e does not grow from Kyber512 to Kyber1024 (e.g. *pqm4* at -00: $e = 10, 8, 8$; reference at -00: $e = 11, 10, 10$), and the dominant variation is instead *across* optimisation levels and code bases. This is precisely where the compiler re-emits the two/three-instruction Booleanisation and the surrounding trigger stores.

We stress that this is a property of the measurement harness, not of the attack. Had the trigger been placed at decapsulation entry rather than at the comparison, e would have absorbed the k -dependent comparison time and grown with the security level; the sensitive Booleanisation is the same short instruction sequence in every case. The practical consequence for a reproducing attacker is the one reflected in Table 4: fix (w, o) once for the rig, then re-profile the small e per build, because it is the compiler’s placement of the Booleanisation, not the parameter set, that moves the target in time.

6.2 Deployment considerations

Reset/crash rate and device-side protections. Table 3 shows that most injections do not yield the intended skip: for Kyber512 only 1320 of 24,195 executions skipped, while 12,647 reset or crashed the target and 10,228 ran normally and rejected. Two consequences follow. First, in our threat model (physical possession, unrestricted invocation, and control of the power and reset lines) these non-successful executions cost only time: the harness detects a crash by the absence of the capture trigger, resets the target, and retries, and the wall-clock totals in the table already include this overhead. Second, a deployed device could exploit exactly this signal. A monotonic fault or failed-decapsulation counter held in non-volatile memory, with key invalidation or a lockout once a threshold is exceeded, would be tripped thousands of times over during a full key recovery and would stop the attack. Three caveats keep this from being a complete answer. (i) The counter must persist across resets: the attacker controls power and reset, so a volatile counter is cleared by the very crash it should record. (ii) Implicit rejection deliberately makes an unfaulted rejection indistinguishable from an accept at the protocol level, so a naive “failed decapsulation” counter would also penalise honest invalid ciphertexts; a counter on *detected faults* (from a watchdog or brown-out reset, or from the integrity check of Section 7.1) is the appropriate trigger. (iii) Such mechanisms are system-level policies outside the Kyber implementation, and none of the evaluated code bases provides one; we therefore regard them as complementary to, not a substitute for, hardening the comparison itself.

Remote feasibility. The primitive is a clock glitch: it requires electrical access to the target’s clock line, or the ability to substitute the clock source. It is therefore not a remote attack and cannot be mounted over a network interface against a cloud or server deployment. Software-controllable fault channels such as voltage/frequency manipulation or memory disturbance are different primitives with different timing precision; whether they can reproduce a sub-instruction skip on the Booleanisation is an open question that we do not claim to answer. The attack’s natural targets are embedded and IoT devices in an adversary’s physical possession, which is the setting in which Cortex-M4 Kyber deployments and the *pqm4*/PQClean code bases are used.

Why the skip-success rate varies across code bases. Table 4 shows that at a fixed intensity (w, o) the per-injection skip-success rate $s\%$ differs markedly by code base: *pqm4* stays around 5–28%, whereas several reference and PQClean builds at *-01--03* reach 40–58%. The target sequence is the same in all cases, so we attribute the difference to the instruction context around it, namely the alignment of the *negs/sbc.w* pair relative to the fetch boundary and the density of the surrounding code, both of which affect how cleanly a single perturbed clock edge lands on the intended instruction. We have not isolated these factors experimentally and state this as an attribution rather than a demonstrated cause.

7 Mitigation

Default Correct succeeds because the entire accept/reject decision rests on a *single* Booleanised flag computed once: forcing that one value (or, equivalently, the default-fail *cmov* that consumes it) is enough to accept any ciphertext. A robust mitigation therefore cannot leave the Booleanisation as a single unguarded flag: as our results show, any reduction to one flag whose execution is not itself verified can be forced by one well-placed skip. Below we first outline the generic directions that raise the bar and remain consistent with our threat model, and then present and evaluate a lightweight producer-side countermeasure, an integrity-checked Booleanisation that detects any partial skip of the reduction at negligible cost (Section 7.1).

Redundancy and detection. Computing the comparison and its Booleanisation *twice*, ideally with diverse code or operand encodings, and aborting on disagreement turns a single-fault accept into a detectable inconsistency: a lone **Default Correct** skip sets one instance to *fail* = 0 while the other still reports *fail* = 1. Double decapsulation / double execution and redundant comparison [BDH⁺21] fall in this class. Defeating them requires *two* correlated faults per query, with success probability of the order of the square of the single-skip rate.

Infective computation. Instead of branching on *fail*, the implicit-rejection secret *z* can be diffused into the released key so that any mismatch unconditionally randomises the output, leaving no “*fail* = 1 path” to skip. An attacker must then forge the entire comparison value rather than flip one flag.

Hardened Booleanisation. Replacing the two’s-complement *negs/sbc.w/lsrcs* sequence with a reduction that verifies its own execution removes the specific single-instruction target we exploit *and* provides detection. We present and evaluate such a replacement next.

7.1 An Integrity-Checked Booleanisation Countermeasure

Design. The idea is to compute the mismatch flag and an execution counter in the *same* instructions, so that no instruction can be skipped without disturbing both. Fig. 11 gives the routine, *ic_booleanise*. It is a drop-in replacement for the two’s-complement idiom. Its return value is a one-byte flag, which we denote *b* and which takes the place of *fail*: *b* = 0 on a fault-free match, *b* = 1 on a fault-free mismatch, and *b* $\notin \{0, 1\}$ whenever execution deviated. The state *os* packs two fields: bit 0 accumulates the OR of the bits of the low byte of *rc*, and the upper nibble holds a counter incremented once per step. Each loop step performs both updates in a single expression, and the return statement applies the final increment and XORs out the expected counter value (9, i.e. 0x90). A fault-free run therefore cancels the counter exactly and leaves only the mismatch bit.

```

/* Replaces rc = ~rc + 1; rc >>= 63; (integrity-checked) */
uint8_t ic_booleanise(uint64_t rc) {
    uint8_t os = (rc & 1) | 0x10; /* counter=1, bit0=rc[0] */
    for (int i = 1; i < 8; i++) { /* unrolled when deployed */
        rc >>= 1;
        os = (((os >> 4) + 1) << 4) | (os & 1) | (rc & 1);
    } /* counter == 8 if fault-free */
    uint8_t b = (((os >> 4) + 1) << 4) | (os & 1) ^ 0x90;
    return b; /* b: 0 match, 1 mismatch, else fault */
}

```

Figure 11: The integrity-checked Booleanisation *ic_booleanise*, returning the flag *b*. The counter increment and the mismatch-OR share one expression on every step, so a skipped step disturbs both; the final XOR cancels the counter only if all nine increments executed.

Fault-free correctness. For every value of the low byte of *rc*, the routine returns exactly the two’s-complement flag: *b* = 0 if and only if the byte is 0. Bits above bit 7 are ignored, which is correct because all three code bases OR byte-wise differences into the accumulator, so every mismatch is visible in its low byte (*verify()* even declares the accumulator as a *uint8_t*, Fig. 6).

Why a skip is detected. The counter increment and the mismatch-OR are evaluated in a single expression, and in our -O3 builds the compiler emits each step’s update as a single instruction. A skip therefore removes a whole update, never one half of it. Skipping the update on any step drops one increment, so the counter reaches at most 8 rather than 9 and the final XOR leaves a non-zero upper nibble; the same holds for skipping the initialisation or the return-time fold. Crucially, an attacker cannot drop a mismatch bit without also dropping an increment, and cannot repair the counter without a second, compensating fault. The argument is exact. If the final fold executes, $b = 0$ requires the counter to reach 8 before the fold, which requires the initialisation and all seven updates to have run, i.e. a fault-free execution whose flag bit is 1 on any mismatch. If the final fold is skipped, b is the raw state, and $b = 0$ requires a counter of 0, i.e. neither the initialisation nor any update ran. We confirmed this exhaustively in a bit-accurate model in which each update is skipped as a unit, matching the compiled code: over all 255 non-zero low bytes of `rc` and all $2 \times 2^7 \times 2 = 512$ combinations of skipping the initialisation, any subset of the seven updates, and the final fold, the *only* combination that returns $b = 0$ on a genuine mismatch is the one that skips every instruction of the routine, i.e. removes it entirely. Every partial skip is detected. Bypassing the check therefore requires faulting the whole routine, a correlated multi-fault attack well outside the single-glitch model of this paper.

Integration. The consumer must treat *any* non-zero b as a rejection (or as a detected fault to be handled by policy), rather than computing $1 - \text{fail}$ on the assumption that $\text{fail} \in \{0, 1\}$. In the default-fail `cmov` this is a one-line change: derive the selection mask from $[b = 0]$ instead of from $1 - b$. Detected faults are also the right signal for a persistent fault counter (Section 6.2).

Overhead and experimental status. We implemented the routine in place of the two’s-complement reduction and measured the decapsulation cycle count on the STM32F303. The added cost is negligible: well under 0.01% of decapsulation cycles for Kyber512, Kyber768, and Kyber1024 in the -O3 build of the *pqm4* library. The routine replaces a three-instruction sequence with a few dozen instructions, while decapsulation is dominated by the NTT and hashing. Clock glitching the hardened build with the parameters of Table 4 did not bypass the comparison in our experiments; the injected skips and corruptions were detected as non-zero b .

8 Conclusion

We have shown that Kyber’s FO ciphertext comparison is not an atomic check but a short pipeline. Its two’s-complement *Booleanisation* is an independently faultable stage *upstream* of the conditional move. A single sub-instruction clock glitch forces the comparison flag *fail* to 0. We call this the **Default Correct** fault. Decapsulation then accepts any ciphertext, valid or invalid. The default-fail `cmov` of Xagawa *et al.* is the only countermeasure proposed for this comparison, and it protects the consumer of the flag. The faulted stage feeds that consumer rather than being protected by it. The fault therefore succeeds on all 36 hardened configurations we evaluate, across the *pqm4*, NIST reference, and PQClean implementations of Kyber-512/768/1024. We drive the resulting accept-everything behaviour with the standard plaintext-checking oracle. The device output is matched against keys precomputed for each candidate message, which is legal since $\mathcal{G}$ and $\mathcal{H}$ are public. In this way we recover the full secret key of all three parameter sets, each in a single measured on-device run.

The fault targets a stage upstream of the `cmov`, so no defence confined to the conditional move can address it. Hardening a single stage of a multi-stage comparison leaves the other stages exposed. Redundancy, double execution, and infective computation raise the bar, and so does our integrity-checked Booleanisation at the producer itself. Each

of these forces an attacker into two or more correlated faults per query. The integrity-checked Booleanisation replaces the vulnerable two's-complement reduction with a self-verifying equivalent. It detects every partial skip at negligible cycle overhead. A full multi-fault evaluation of these defences is left to future work. The *mkm4* [HKL⁺22] and *pqm4_masked* [SIM22, BC22] libraries retain the producer and consumer stages, but their producers do not use the two's-complement Booleanisation and the *lsrs* shift. They realise the comparison predicate through a masked share-wise comparison followed by a distinct recombination and compression step. A skip analysis of these producer stages, or of their equivalents, remains future work, as does the study of the fault against duplicated implementations.

A Oracle threshold construction: isolation and feasibility

This appendix derives the conditions under which the crafted ciphertext of Section 3.2 makes the oracle $\mathcal{O}(V_c, U_c)$ equal the comparator $[s_i \geq c]$, and reports their exhaustive verification. We use the standard Kyber parameters $q = 3329$ and, per level, $(\eta_1, d_u, d_v) = (3, 10, 4)$ for Kyber512, $(2, 10, 4)$ for Kyber768, and $(2, 11, 5)$ for Kyber1024. The device decompresses the crafted scalars to

$$\tilde{u} = \text{Decompress}_q(U, d_u) = \left\lfloor \frac{q}{2^{d_u}} U \right\rfloor, \quad \tilde{v} = \text{Decompress}_q(V, d_v) = \left\lfloor \frac{q}{2^{d_v}} V \right\rfloor,$$

and decodes each coefficient with $\text{Decode}(w) = 1 \iff (w \bmod q) \in [q/4, 3q/4)$. Because the attacker supplies the compressed ciphertext directly, $\tilde{u}$ and $\tilde{v}$ are exact and the oracle is deterministic.

Decoded target bit. With the sparse ciphertext of Eq. (2) (the u -polynomial set to the single monomial $\tilde{u} X^{256-i}$ and v to the constant $\tilde{v}$), the negacyclic product contributes $-\tilde{u} s_i$ to the constant coefficient (Eq. (3)), so for $i \in \{1, \dots, 255\}$

$$w[0] = \tilde{v} + \tilde{u} s_i \pmod{q}, \quad (7)$$

which is monotonically increasing in s_i . (For $i = 0$ the u -monomial is $\tilde{u} X^0$, whose product term does not wrap modulo $X^{256} + 1$, so the sign flips: $w[0] = \tilde{v} - \tilde{u} s_0$, and the inequalities below are mirrored; see the discussion following Eq. (3).)

Isolation. We first argue that only position 0 can carry a nonzero message bit, by writing out the decryption product explicitly. Decapsulation forms the polynomial $w = v - u s$ in the negacyclic ring $R_q = \mathbb{Z}_q[X]/(X^{256} + 1)$. With the crafted scalars $v = \tilde{v}$ (a constant) and $u = \tilde{u} X^{256-i}$, expanding against $s = \sum_{j=0}^{255} s_j X^j$ gives

$$u s = \tilde{u} \sum_{j=0}^{255} s_j X^{256-i+j}.$$

Each exponent $256 - i + j$ lies in $[256 - i, 511 - i]$; reducing modulo $X^{256} + 1$ uses $X^{256+m} = -X^m$, so the term from index j lands on output position $(j - i) \bmod 256$ with sign -1 exactly when it wraps (i.e. when $j \geq i$). Output position t therefore collects the *single* secret index $(t + i) \bmod 256$, giving $(u s)[t] = \pm \tilde{u} s_{(t+i) \bmod 256}$. Since v touches only position 0, subtracting yields $w[0] = \tilde{v} + \tilde{u} s_i$ (Eq. (7)) and, for every $t \neq 0$,

$$w[t] = \mp \tilde{u} s_{(i+t) \bmod 256}, \quad t \neq 0. \quad (8)$$

The isolation argument now proceeds in two steps. (i) *Bound the magnitude.* Every secret coefficient obeys $|s_j| \leq \eta_1$ (the CBD range), so from Eq. (8) $|w[t]| \leq \tilde{u} \eta_1$. (ii) *Force it*

below the decode band. **Decode** returns 1 only on the band $[q/4, 3q/4)$; a coefficient with $|w[t]| < q/4$ therefore lies in $(-q/4, q/4) \bmod q$ and decodes to 0. Combining (i) and (ii), it suffices that the worst case $\tilde{u}\eta_1$ itself stays under $q/4$:

$$\tilde{u}\eta_1 < q/4. \quad (9)$$

Under Eq. (9) every $w[t]$ with $t \neq 0$ decodes to 0, so the decapsulated message is zero everywhere except possibly at position 0 and the oracle output equals the single bit $\text{Decode}(w[0])$.

Comparator. We now pick $\tilde{v}$ so that this surviving bit reads $[s_i \geq c]$. Recall $w[0] = \tilde{v} + \tilde{u}s_i$ (Eq. (7)), monotone in s_i , and $\text{Decode}(w[0]) = 1$ iff $w[0] \in [q/4, 3q/4)$. We impose the two edges of this band in turn. (i) *Lower edge: place the threshold at c .* Keep $w[0]$ below the upper boundary for now (step (ii) enforces this), so $\text{Decode}(w[0]) = 1$ reduces to crossing the lower boundary: $w[0] \geq q/4$. Substituting $w[0] = \tilde{v} + \tilde{u}s_i$ and solving for s_i (with $\tilde{u} > 0$),

$$\tilde{v} + \tilde{u}s_i \geq \frac{q}{4} \iff s_i \geq \frac{q/4 - \tilde{v}}{\tilde{u}} =: \tau.$$

Because s_i is an integer, $[s_i \geq \tau]$ equals $[s_i \geq c]$ precisely when the real threshold τ falls in the half-open unit interval $(c-1, c]$, i.e. $c-1 < \tau \leq c$. Writing this out and multiplying through by $\tilde{u} > 0$,

$$\tilde{u}(c-1) < q/4 - \tilde{v} \leq \tilde{u}c,$$

then subtracting $q/4$ and negating (which flips the inequalities) gives the first condition of

$$\frac{q}{4} - \tilde{u}c \leq \tilde{v} < \frac{q}{4} - \tilde{u}(c-1), \quad \tilde{v} + \tilde{u}\eta_1 \leq \frac{3q}{4}. \quad (10)$$

(ii) *Upper edge: do not overshoot the band.* $w[0]$ is largest at $s_i = \eta_1$; requiring that this maximum stay inside $[q/4, 3q/4)$ (so a true answer never wraps past the upper boundary and flips back to 0) gives $\tilde{v} + \tilde{u}\eta_1 \leq 3q/4$, the second condition in Eq. (10). Together the two edges yield $\text{Decode}(w[0]) = 1 \iff s_i \geq c$. The named oracles of Fig. 1 are thus $\mathcal{O}(V_{>0}, U_{>0}) = [s_i \geq 1]$, $\mathcal{O}(V_{+2}, U_{+2}) = [s_i \geq 2]$, $\mathcal{O}(V_{+3}, U_{+3}) = [s_i \geq 3]$, and the negative tests are their complements, $\mathcal{O}(V_{<0}, U_{<0}) = \neg[s_i \geq 0]$, $\mathcal{O}(V_{-2}, U_{-2}) = \neg[s_i \geq -1]$, $\mathcal{O}(V_{-3}, U_{-3}) = \neg[s_i \geq -2]$.

Feasibility. It remains to show that a legal compressed pair (U, V) realising Eq. (10) actually exists at every level. The only free integer is U (which fixes $\tilde{u} \approx \frac{q}{2^{d_u}}U$); we bound it from two sides and then check the window is non-empty. (i) *Lower bound: $\tilde{v}$ must be reachable.* The interval for $\tilde{v}$ in Eq. (10) has width exactly $\tilde{u}$. But $\tilde{v}$ is not free: it is a decompressed value and so lands on a grid of spacing $q/2^{d_v}$ (one step of V moves $\tilde{v}$ by $q/2^{d_v}$). The interval contains a grid point, hence an attainable $\tilde{v}$, as soon as its width is at least the spacing, $\tilde{u} \geq q/2^{d_v}$. Substituting $\tilde{u} = \frac{q}{2^{d_u}}U$ and cancelling q gives $U \geq 2^{d_u-d_v}$. (ii) *Upper bound: isolation.* The isolation condition Eq. (9), $\tilde{u}\eta_1 < q/4$, becomes after the same substitution $\frac{q}{2^{d_u}}U\eta_1 < q/4$, and cancelling q gives $U < 2^{d_u}/(4\eta_1)$. Stacking (i) and (ii), and noting that q has dropped out of both, the compressed amplitude must satisfy

$$2^{d_u-d_v} \leq U < \frac{2^{d_u}}{4\eta_1}, \quad (11)$$

which is non-empty for every level: $64 \leq U \leq 85$ for Kyber512 ($2^6 = 64$, $2^{10}/12 = 85.3$), $64 \leq U \leq 127$ for Kyber768 ($2^{10}/8 = 128$), and $64 \leq U \leq 255$ for Kyber1024 ($2^6 = 64$, $2^{11}/8 = 256$). Table 5 gives a concrete instantiation (a single U per level and one V per threshold). We verified exhaustively, over all $s_i \in [-\eta_1, \eta_1]$ and worst-case as well as random

assignments of the remaining coefficients, that each entry reproduces its comparator and preserves isolation.

Table 5: Concrete oracle instantiation realising $[s_i \geq c]$ (target index $i = 1$): one compressed amplitude U per level and the compressed offset V per threshold. Dashes mark thresholds outside the secret range. All entries are verified exhaustively over every secret value, as described above.

	Kyber512 ($U=64, \tilde{u}=208$)	Kyber768 ($U=64, \tilde{u}=208$)	Kyber1024 ($U=64, \tilde{u}=104$)
Comparator	V	V	V
$[s_i \geq -2]$	7	—	—
$[s_i \geq -1]$	6	6	10
$[s_i \geq 0]$	5	5	9
$[s_i \geq +1]$	4	4	8
$[s_i \geq +2]$	3	3	7
$[s_i \geq +3]$	2	—	—

References

- [AAC⁺22] Gorjan Alagic, Daniel Apon, David Cooper, Quynh Dang, Thinh Dang, John Kelsey, Jacob Lichtinger, Yi-Kai Liu, Carl Miller, Dustin Moody, Rene Peralta, Ray Perlner, Angela Robinson, and Daniel Smith-Tone. Status report on the third round of the nist post-quantum cryptography standardization process. Technical Report NIST IR 8413, National Institute of Standards and Technology (NIST), 2022. <https://doi.org/10.6028/NIST.IR.8413>.
- [BC22] Olivier Bronchain and Gaëtan Cassiers. Bitslicing arithmetic/boolean masking conversions for fun and profit with application to lattice-based KEMs. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2022(4):553–588, 2022.
- [BDH⁺21] Shivam Bhasin, Jan-Pieter D’Anvers, Daniel Heinz, Thomas Pöppelmann, and Michiel Van Beirendonck. Attacking and defending masked polynomial comparison for lattice-based cryptography. In *IACR Transactions on Cryptographic Hardware and Embedded Systems (TCHES)*, 2021. <https://tches.iacr.org/index.php/TCHES/article/view/8799>.
- [BDL01] Dan Boneh, Richard A DeMillo, and Richard J Lipton. On the importance of eliminating errors in cryptographic computations. *Journal of Cryptology*, 14(2):101–119, 2001. <https://doi.org/10.1007/s001450010010>.
- [BP18] Leon Groot Bruinderink and Peter Pessl. Differential fault attacks on deterministic lattice signatures. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 21–43, 2018. <https://tches.iacr.org/index.php/TCHES/article/view/837>.
- [BS97] Eli Biham and Adi Shamir. Differential fault analysis of secret key cryptosystems. In *Annual International Cryptology Conference*, pages 513–525, 1997. <https://doi.org/10.1007/BFb0052259>.
- [CS03] Ronald Cramer and Victor Shoup. Design and analysis of practical public-key encryption schemes secure against adaptive chosen ciphertext attack. *SIAM Journal on Computing*, 33(1):167–226, 2003. <https://doi.org/10.1137/S0097539702405288>.

- [Del22] Jeroen Delvaux. Roulette: A diverse family of feasible fault attacks on masked Kyber. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2022(4):637–660, 2022.
- [Den03] Alexander W Dent. A designer’s guide to KEMs. In *IMA International Conference on Cryptography and Coding*, pages 133–151, 2003. https://doi.org/10.1007/3-540-36288-6_10.
- [EFGT17] Thomas Espitau, Pierre-Alain Fouque, Benoît Gérard, and Mehdi Tibouchi. Side-channel attacks on BLISS lattice-based signatures: Exploiting branch tracing against strongswan and electromagnetic emanations in microcontrollers. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 1857–1874, 2017. <https://doi.org/10.1145/3133956.3134017>.
- [ESH⁺11] Sho Endo, Takeshi Sugawara, Naofumi Homma, Takafumi Aoki, and Akashi Satoh. An on-chip glitchy-clock generator for testing fault injection attacks. *Journal of Cryptographic Engineering*, 1(4):265–270, 2011. <https://doi.org/10.1007/s13389-011-0020-2>.
- [FO99] Eiichiro Fujisaki and Tatsuaki Okamoto. Secure integration of asymmetric and symmetric encryption schemes. In *Annual international cryptology conference*, pages 537–554. Springer, 1999.
- [FO13] Eiichiro Fujisaki and Tatsuaki Okamoto. Secure integration of asymmetric and symmetric encryption schemes. *Journal of cryptology*, 26(1):80–101, 2013.
- [GBHLY16] Leon Groot Bruinderink, Andreas Hülsing, Tanja Lange, and Yuval Yarom. Flush, gauss, and reload—a cache attack on the bliss lattice-based signature scheme. In *International Conference on Cryptographic Hardware and Embedded Systems*, pages 323–345, 2016. https://link.springer.com/chapter/10.1007/978-3-662-53140-2_16.
- [GJN19] Qian Guo, Thomas Johansson, and Alexander Nilsson. A key-recovery timing attack on post-quantum primitives using the fujisaki-okamoto transformation and its application on frodokem. In *Advances in Cryptology – CRYPTO 2019*, pages 3–33. Springer, 2019. https://doi.org/10.1007/978-3-030-26951-7_1.
- [HHK17] Dennis Hofheinz, Kathrin Hövelmanns, and Eike Kiltz. A modular analysis of the Fujisaki–Okamoto transformation. In Yael Kalai and Leonid Reyzin, editors, *Theory of Cryptography – TCC 2017*, volume 10677 of *Lecture Notes in Computer Science*, pages 341–371. Springer, 2017. https://doi.org/10.1007/978-3-319-70500-2_12.
- [HHS⁺11] Yu-ichi Hayashi, Naofumi Homma, Takeshi Sugawara, Takaaki Mizuki, Takafumi Aoki, and Hideaki Sone. Non-invasive trigger-free fault injection method based on intentional electromagnetic interference. In *Non-Invasive Attack Testing Workshop*, 2011. https://link.springer.com/chapter/10.1007/978-3-642-25288-8_4.
- [HKL⁺22] Daniel Heinz, Matthias J. Kannwischer, Georg Land, Thomas Pöppelmann, Peter Schwabe, and Amber Sprenkels. First-order masked Kyber on ARM Cortex-M4. Cryptology ePrint Archive, Paper 2022/058, 2022. <https://github.com/masked-kyber-m4/mkm4>.

- [HPP21] Julius Hermelink, Peter Peßl, and Thomas Pöppelmann. Fault-enabled chosen-ciphertext attacks on Kyber. In *Progress in Cryptology – INDOCRYPT 2021*, volume 13143 of *LNCS*, pages 311–334. Springer, 2021. <https://eprint.iacr.org/2021/1222>.
- [KRSS19] Matthias J. Kannwischer, Joost Rijneveld, Peter Schwabe, and Ko Stoffelen. pqm4: Testing and benchmarking NIST PQC on ARM Cortex-M4. Workshop Record of the Second PQC Standardization Conference; Cryptology ePrint Archive, Paper 2019/844, 2019. <https://github.com/mupq/pqm4>.
- [Nat16] National Institute of Standards and Technology (NIST). Submission Requirements and Evaluation Criteria for the Post-Quantum Cryptography Standardization Process. Technical report, National Institute of Standards and Technology, December 2016. <https://csrc.nist.gov/csrc/media/projects/post-quantum-cryptography/documents/call-for-proposals-final-dec-2016.pdf>.
- [Nat24] National Institute of Standards and Technology. Module-Lattice-Based Key-Encapsulation Mechanism Standard. Federal Information Processing Standards Publication (FIPS) 203, U.S. Department of Commerce, 2024. <https://doi.org/10.6028/NIST.FIPS.203>.
- [New] NewAE Technology Inc. Chipwhisperer-lite capture hardware documentation. <https://rtfm.newae.com/Capture/ChipWhisperer-Lite/>. Accessed: 2026.
- [Pay20] Payatu. Glitching on stm32 blue pill, 2020. <https://payatu.com/blog/glitching-on-stm32-blue-pill/>.
- [Pes16] Peter Pessl. Analyzing the shuffling side-channel countermeasure for lattice-based signatures. In *International Conference on Cryptology in India*, pages 153–170, 2016. https://link.springer.com/chapter/10.1007/978-3-319-49192-5_9.
- [PP19] Peter Pessl and Robert Primas. More practical single-trace attacks on the number theoretic transform. Cryptology ePrint Archive, Paper 2019/795, 2019.
- [PP21] Peter Pessl and Lukas Prokop. Fault attacks on cca-secure lattice kems. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2021(2):37–60, Feb 2021. <https://doi.org/10.46586/tches.v2021.i2.37-60>.
- [QCZ⁺21] Yue Qin, Chi Cheng, Xiaohan Zhang, Yanbin Pan, Lei Hu, and Jintai Ding. A systematic approach and analysis of key mismatch attacks on lattice-based NIST candidate KEMs. In *Advances in Cryptology – ASIACRYPT 2021*, volume 13092 of *LNCS*, pages 92–121. Springer, 2021. <https://eprint.iacr.org/2021/123>.
- [RBRB⁺18] 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 *Cryptographic Hardware and Embedded Systems – CHES 2018*, pages 65–85. Springer, 2018. https://doi.org/10.1007/978-3-319-98989-7_3.
- [RBRC20] Prasanna Ravi, Shivam Bhasin, Sujoy Sinha Roy, and Anupam Chattopadhyay. Drop by Drop you break the rock-Exploiting generic vulnerabilities in Lattice-based PKE/KEMs using EM-based Physical Attacks. *IACR Cryptology ePrint Archive*, 2020:549, 2020. <https://eprint.iacr.org/2020/549>.

- [RCDB22] Prasanna Ravi, Anupam Chattopadhyay, Jan-Pieter D’Anvers, and Anubhab Baksi. Side-channel and fault-injection attacks over lattice-based post-quantum schemes (kyber, dilithium): Survey and new results. *ACM Transactions on Embedded Computing Systems*, 21(6):1–36, 2022.
- [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*, 2020(3):307–335, 2020. <https://doi.org/10.13154/tches.v2020.i3.307-335>; ePrint 2019/948.
- [Sho94a] Peter W Shor. Algorithms for quantum computation: discrete logarithms and factoring. In *Proceedings 35th Annual Symposium on Foundations of Computer Science*, pages 124–134, 1994. <https://doi.org/10.1109/SFCS.1994.365700>.
- [Sho94b] Peter W Shor. Polynomial time algorithms for discrete logarithms and factoring on a quantum computer. In *International Algorithmic Number Theory Symposium*, pages 289–289, 1994. <https://doi.org/10.1007/BF02355492>.
- [SIM22] SIMPLE-Crypto. pqm4_masked: Masked implementations of post-quantum schemes for the ARM Cortex-M4. https://github.com/simple-crypto/pqm4_masked, 2022.
- [SMR09] Dhiman Saha, Debdeep Mukhopadhyay, and Dipanwita RoyChowdhury. A diagonal fault attack on the advanced encryption standard. *Cryptology ePrint Archive*, 2009. <https://eprint.iacr.org/2009/581>.
- [SS93] Gordon B. Steven and Fleur L. Steven. Alu design and processor branch architecture. *Microprocessing and Microprogramming*, 36(5):259–278, 1993. [https://doi.org/10.1016/0165-6074\(93\)90264-L](https://doi.org/10.1016/0165-6074(93)90264-L).
- [Tea22] Tristen Teague. Side-Channel Analysis on Post-Quantum Cryptography Algorithms. Undergraduate honors thesis, University of Arkansas, Fayetteville, May 2022. Available at ScholarWorks@UARK.
- [The21] The FrodoKEM Team. FrodoKEM: Practical quantum-secure key encapsulation from generic lattices (reference implementation). <https://github.com/microsoft/PQCrypto-LWEKE>, 2021. NIST PQC Round 3 submission.
- [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*, 2022(1):296–322, 2022. <https://doi.org/10.46586/tches.v2022.i1.296-322>.
- [WCCL25] Jian Wang, Weiqiong Cao, Hua Chen, and Haoyuan Li. Secret in One-Piece: Single-bit fault attack on Kyber. In *Selected Areas in Cryptography – SAC 2025*, 2025. <https://sacworkshop.org/SAC25/preproceedings/sac2025-1-paper1.pdf>. LNCS volume/pages pending final proceedings.
- [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 Mehdi Tibouchi and Huaxiong Wang, editors, *Advances in Cryptology – ASIACRYPT 2021*, pages 33–61, Cham, 2021. Springer International Publishing. https://doi.org/10.1007/978-3-030-92075-3_2.