Exploring Fault Injection Attacks on HQC and BIKE Beyond Fujisaki-Okamoto Verification

Thales B. Paiva[1], Lucas C. Cardoso[1], Keerthi Kamalakshan[2,3], Hariprasad Kelassery Valsaraj[3,4], Marcos A. Simplicio Jr.[1] and Shivam Bhasin[2,3]

[1] Universidade de São Paulo, São Paulo, Brazil
{thalespaiva,lcardoso,mjunior}@larc.usp.br
[2] Temasek Laboratories, Nanyang Technological University, Singapore
[3] National Integrated Centre For Evaluation, Nanyang Technological University, Singapore
{keerthi.kamalakshan,sbhasin}@ntu.edu.sg,HARIPRAS003@e.ntu.edu.sg
[4] School of Physical and Mathematical Sciences, Nanyang Technological University, Singapore

Abstract. Fault injection attacks have been extensively studied in the context of post-quantum cryptography, yet existing analyses on code-based KEMs largely focus on faults targeting the Fujisaki-Okamoto transform. This work presents new fault injection attacks against BIKE and HQC that operate beyond this commonly protected component and target less obvious instructions within the decapsulation process. The proposed attacks achieve full key recovery using only valid ciphertexts under NIST Level 1 parameters, requiring 17392 and approximately 3 million decapsulation queries for HQC and BIKE, respectively. In addition, we introduce a general approach to identify effective fault locations by analyzing decoder behavior under perturbed conditions. The attacks are experimentally validated on an STM32H7 microcontroller using laser fault injection, demonstrating high repeatability. These results reveal a broader attack surface and provide a foundation for more efficient fault attack analyses in future work.

Keywords: HQC · BIKE · Fault Injection · Fujisaki-Okamoto Transform

1 Introduction

In 2016, the National Institute of Standards and Technology (NIST) launched a standardization process for post-quantum secure key encapsulation mechanisms (KEMs) and digital signature schemes. At the end of the third round in 2022, the lattice-based scheme CRYSTALS-Kyber [ABD+21] was the first KEM to be selected for standardization, which was later published as ML-KEM [Nat24]. At that time, three code-based KEMs were still being considered for standardization: BIKE [ABB+24], HQC [GAMA+25], and Classic McEliece [ABC+22]. In 2025, NIST selected HQC to be standardized due to its public keys being significantly shorter than those of Classic McEliece, and because it provides a more mature analysis of the decryption failure rate (DFR) when compared to BIKE.

To achieve IND-CCA security, these KEMs are built from an underlying IND-CPA public-key encryption scheme using a variant of the Fujisaki-Okamoto (FO) transform [HHK17, FO13], typically the implicit rejection variant. The core idea of the FO transform is to split the decapsulation of a ciphertext $\mathbf{c}$ into three steps: (*i*) decrypt $\mathbf{c}$ to obtain $\mathbf{m}$, (*ii*) deterministically re-encrypt $\mathbf{m}$ to obtain $\mathbf{c}'$, and (*iii*) check whether $\mathbf{c} = \mathbf{c}'$. If $\mathbf{c} = \mathbf{c}'$, decapsulation derives the real key via a key derivation function (KDF), $K \leftarrow \mathrm{KDF}(\mathbf{m}, \mathbf{c})$. On the other hand, if $\mathbf{c} \neq \mathbf{c}'$ or a decryption failure occurred, decapsulation performs *implicit rejection*, returning a *fake* key $K_{\mathrm{rej}} \leftarrow \mathrm{KDF}(\mathbf{c}, z)$, where z is a fixed secret value held by the receiver.

Implicit rejection does not, however, conceal whether the receiver derived the same key as the sender: if the receiver derives K_{rej} (unknown to the sender), subsequent communication using the shared key can fail (e.g. an AEAD ciphertext may be rejected as invalid), and the sender may observe this as a decapsulation failure. When the FO check succeeds, it confirms that the sender's ciphertext was honestly generated; when it fails, implicit rejection hides *why* it failed, i.e., whether decryption failed or an invalid ciphertext was detected.

This is what allows the FO transform to prevent reaction attacks [HGS99] using invalid ciphertexts: a class of attacks that exploit correlations between decryption failures and the secret key. Since decapsulation returns K_{rej} both when an invalid ciphertext is detected and when decryption fails, an attacker using invalid ciphertexts cannot distinguish between the two cases. If the attacker instead submits *valid* ciphertexts, however, a decapsulation failure can only be due to a decryption failure, and this correlation can be exploited for reaction attacks [FHS+17, PT18, SSPB19, GJ20]. For this reason, modern code- and lattice-based schemes are designed for negligible decryption failure rate (DFR), so failures on valid ciphertexts should essentially never occur.

Due to the importance of the FO verification in thwarting reaction attacks, it is a natural target for fault injection: skipping the verification reopens reaction attacks using invalid ciphertexts, since the negligible DFR guarantee applies only to valid ciphertexts. Xagawa et al. [XIU+21] systematically showed that a single instruction-skipping fault suffices to bypass the FO equality check across all NIST Round 3 KEM candidates with PQM4 implementations.

Beyond this generic single-fault result, a growing body of work has targeted specific schemes with tailored fault attacks. For lattice-based schemes, several such attacks have been demonstrated. Ravi et al. [RRB+19] demonstrated the first practical fault attacks on lattice-based schemes, showing that nonce-manipulation faults enable key and message recovery. Pessl and Prokop [PP21] demonstrated full key recovery on Kyber512, requiring approximately 6500 faulty decapsulations. Subsequent work by Hermelink et al. [HPP21], Delvaux [Del22], and Hermelink et al. [HMS+23] progressively widened the attack surface, relaxed fault-model requirements, and improved key-recovery efficiency through the combination of belief propagation and lattice reduction. Beyond scheme-specific operations, Kelassery Valsaraj et al. [VRBW25] identified a single point of failure in the random seed pointer used for polynomial sampling in both ML-KEM and ML-DSA, using fault injection.

The comprehensive survey of Ravi et al. [RCDB24] on implementation attacks against lattice-based schemes have no direct counterpart for code-based schemes, reflecting a much sparser body of published work. In particular, Classic McEliece was targeted by fault attacks [CCD+21, PGD+23], but for BIKE and HQC the only prior fault-attack study remains that of Xagawa et al. [XIU+21] discussed above. There, the authors identified BIKE and HQC as theoretically susceptible to key recovery via plaintext-checking oracle attacks *when the FO verification check is skipped*, but no experimental key-recovery attack was demonstrated. Moreover, the FO verification is well known to be critical for security against fault attacks, and is a high-priority target that one would heavily protect when hardening an implementation. This raises the question of whether it is possible to mount fault injection attacks on BIKE and HQC by exploiting other fault locations.

In this work, we present novel fault injection attacks against BIKE and HQC under a stronger and more realistic security model (see Table 1). Unlike prior work, which primarily focuses on faults affecting the Fujisaki-Okamoto (FO) transform, this study considers a broader class of fault targets, including instructions that are less obvious and more difficult to protect. In addition, we introduce a general method to perform fault injection attacks to extract the key from code-base schemes based on quasi-cyclic codes that use sparse binary vectors as secret keys. As such, our methods are applicable to compact

Table 1. Comparison between our results and Xagawa et al.'s [XIU+21] fault injection attacks against HQC and BIKE, when targeting NIST Level 1 security parameters. The total attack time is driven primarily by decapsulation queries. All timings are based on actual measurements from testing on an STM32H7 microcontroller running at 400 MHz.

Reference	Scheme	Targeted operation	Targeted instruction	Attack surface	No. of decapsulations for key recovery	Estimated time for key recovery
[XIU+21]	BIKE	FO-verification	CMP	Skip	Only partially recovers key	-
	HQC	FO-verification	-	-	NA[1]	-
This work	HQC	Polynomial mult.	AND	Skip	17392	51 minutes
	HQC	Vector addition	Branch	Skip	83.3 million	169 days
	BIKE	BGF threshold	XOR	Skip	3 million	2.57 days
	BIKE	BIKE-Flip iteration	Branch	Skip	16.7 million[2]	14.34 days

[1] Xagawa et al. [XIU+21] considered fault attacks on PQM4 [KPR+] implementations, but no corresponding implementation of HQC was available at the time. Moreover, their work focuses on the sources of vulnerabilities in general, not solely on key recovery.
[2] Due to unavailable constant-time BIKE-Flip implementations, our attack was evaluated against an idealized simulation.

code-based schemes such as BIKE and HQC, but not to Classic McEliece, whose structure is substantially different. The source code and datasets used in this work are publicly available at https://github.com/thalespaiva/fault-injection-on-code-kems.

The main contributions are summarized as follows.

1. *Fault attacks beyond FO verification.* The first fault injection attacks on BIKE and HQC that do not rely on faults targeting the Fujisaki-Okamoto transform are presented. While FO verification is commonly regarded as the primary point of protection, since bypassing it compromises CCA security, these attacks demonstrate that key recovery remains possible even when this component is assumed to be secured. In particular, full secret key recovery can be achieved using only valid ciphertexts, requiring 17392 and 3 million decapsulation queries for HQC and BIKE, respectively, under NIST Level 1 parameters.

2. *Experimental validation.* We experimentally validated these faults on an STM32H7 microcontroller using laser fault injection against the reference implementations of BIKE and HQC. Our results show that the induced faults are highly repeatable, demonstrating the practicality and reliability of the proposed attack methodology. We have also validated that the proposed attacks are applicable to widely used public libraries, including PQM4 [KPR+] and LibOQS [SM17], which employ similar implementation structures. Due to the large number of decapsulation queries required by our attacks, after experimentally validating the fault locations and repeatability rates, we performed software-based fault simulations that reproduce the observed real-world behavior. This enables us to provide accurate and reproducible evaluations of the performance of our attacks.

3. *New general method for exploiting fault attacks on code-based KEMs.* We propose a generic approach to guide the identification and exploitation of effective fault locations in BIKE and HQC. The approach is based on characterizing a class of faults that induce decapsulation failures by increasing the Hamming weight of vectors processed by the decoders, namely the BGF decoder in BIKE and the RMRS decoder in HQC. These fault-induced behaviors are then leveraged to recover the secret key using reconstruction techniques based on approximate distance information between non-zero coordinates, commonly referred to as the key distance spectrum.

4. *Improved key reconstruction algorithm.* As a minor contribution, an enhanced randomized key recovery algorithm is proposed, built upon prior work [GJS16, PT20]. This incorporates practical heuristics that increase robustness to noise and uncertainty, enabling successful key recovery with fewer faults. This key reconstruction algorithm

Table 2. BIKE parameters for each security level [ABB+24].

NIST Security	r	w	d	t	BGF $\texttt{Thresh}(\mathbf{s})$ function	DFR
Level 1	12323	142	71	134	$\max(36, \lfloor 0.00697220\,\lvert\mathbf{s}\rvert + 13.5300 \rfloor)$	2^{-128}
Level 3	24659	206	103	199	$\max(52, \lfloor 0.00526500\,\lvert\mathbf{s}\rvert + 15.2588 \rfloor)$	2^{-192}
Level 5	40973	274	137	264	$\max(69, \lfloor 0.00402312\,\lvert\mathbf{s}\rvert + 17.8785 \rfloor)$	2^{-256}

recovers the key within a couple of minutes for all our attacks. In particular, this means that the total attack time is dominated by the decapsulation queries to the device, not the key recovery phase.

Table 1 summarizes the comparison with prior fault injection results on HQC and BIKE under NIST Level 1 parameters. The results highlight that targeting alternative instructions enables practical key recovery, even when commonly protected components are not involved, indicating a broader and less explored attack surface than previously considered. While the number of required decapsulations may appear large compared to prior fault injection studies, an optimized setup can still complete the attack within hours to a few days depending on the platform and injection rate. Although this is higher than typical practical fault injection settings, this work is the first to investigate faults on internal code-based operations that directly lead to full key recovery in these schemes. The proposed approach further provides a basis for improving efficiency and exploring more effective fault strategies in future work.

2 Background

Notation. We use bold uppercase and lowercase letters to represent matrices and vectors, respectively. An n-bit binary vector $\mathbf{v}$ has their entries denoted as $\mathbf{v}[i]$, for $0 \leq i \leq n-1$. The Hamming weight of a binary vector $\mathbf{v}$ is denoted by $|\mathbf{v}|$. The support of a vector $\mathbf{v}$, denoted by $\mathrm{supp}(\mathbf{v})$, is the set of indices of its non-null entries. Binary n-bit vectors are identified with polynomials in $\mathbb{Z}_2[x]/(x^n-1)$. As such, if $\mathbf{u}$ and $\mathbf{v}$ are binary vectors in $\mathbb{Z}_2^n$, we let $\mathbf{u}\cdot\mathbf{v}$ denote their product under $\mathbb{Z}_2[x]/(x^n-1)$. If $\mathbf{v}\in\mathbb{Z}_2^n$, we let $\mathrm{rot}(\mathbf{v}) \in \mathbb{Z}_2^{n\times n}$ denote the circulant matrix generated by $\mathbf{v}$. The circular distance between indexes i and j of an n-bit binary vector $\mathbf{v}$ is denoted by $\mathrm{dist}_n(i,j) = \min\{|i-j|, n-|i-j|\}$. The distance spectrum of a vector is the set of circular distances between each pair of its non-null entries. More formally, the spectrum of a binary vector $\mathbf{v}$ is defined as the set $\sigma(\mathbf{v}) = \{\mathrm{dist}_n(i,j) : i \neq j \text{ and } \mathbf{v}[i] = \mathbf{v}[j] = 1\}$. A distance $d \in \sigma(\mathbf{v})$ has multiplicity m when there are m different pairs of non-null entries of $\mathbf{v}$ whose circular distance is d. If $d \notin \sigma(\mathbf{v})$, we say that distance d has multiplicity 0. We let $x \leftarrow\!\$\ X$ denote that x is sampled uniformly at random from a set X. If the randomness for uniform sampling is taken from an XOF, we write $x \xleftarrow{\texttt{xof}} X$.

2.1 BIKE

Given a security level, BIKE [ABB+24] uses the corresponding parameters r, w, and t taken from Table 2. Parameters r and w will define the family of QC-MDPC codes to be used, while t controls the weight of the error used for encryption. Listing 1 shows BIKE KEM algorithms with the implicit rejection Fujisaki-Okamoto (FO) [FO99, HHK17] transformation, as presented in the latest specification [ABB+24]. The algorithms require three auxiliary functions $\mathcal{H}, \mathcal{K}$, and $\mathcal{L}$ that are modeled as random oracles. Both $\mathcal{K}$ and $\mathcal{L}$ output 256-bit vectors, and $\mathcal{H}$ outputs pairs of vectors $(\mathbf{e}_0, \mathbf{e}_1) \in \mathbb{Z}_2^r$ such that $|\mathbf{e}_0| + |\mathbf{e}_1| = t$.

The security parameters ensure that $w/2$ is odd and that 2 is primitive modulo r, thus, during key generation, both $\mathbf{h}_0$ and $\mathbf{h}_1$ are invertible in $\mathbb{Z}_2[x]/(x^r-1)$. The secret key is

(a) Key generation.

1 **procedure** `BIKE.KeyGen`(No input)
2 $\mathbf{h}_0 \leftarrow\$ \left\{\mathbf{v} \in \mathbb{Z}_2^r : |\mathbf{v}| = w\right\}$
3 $\mathbf{h}_1 \leftarrow\$ \left\{\mathbf{v} \in \mathbb{Z}_2^r : |\mathbf{v}| = w\right\}$
4 $\mathbf{h}_{\text{pub}} \leftarrow \mathbf{h}_1 \cdot \mathbf{h}_0^{-1}$
5 $\mu \leftarrow$ First 256 bits of $\mathbf{h}_{\text{pub}}$
6 $\sigma \leftarrow\$ \mathbb{Z}_2^{256}$
7 $\mathbf{pk} \leftarrow \mathbf{h}_{\text{pub}}$
8 $\mathbf{sk} \leftarrow (\mathbf{h}_0, \mathbf{h}_1, \mu, \sigma)$
9 **return** $(\mathbf{pk}, \mathbf{sk})$
10 **end procedure**

(b) Encapsulation.

1 **procedure** `BIKE.Encaps`($\mathbf{pk}$)
2 $\mathbf{h}_{\text{pub}} \leftarrow \mathbf{pk}$
3 $\mathbf{m} \leftarrow\$ \mathbb{Z}_2^{256}$
4 $(\mathbf{e}_0, \mathbf{e}_1) \leftarrow \mathcal{H}(\mathbf{m}, \pi_\ell(\mathbf{h}))$
5 $\mathbf{c}_0 \leftarrow \mathbf{e}_0 + \mathbf{e}_1 \cdot \mathbf{h}_{\text{pub}}$
6 $\mathbf{c}_1 \leftarrow \mathbf{m} \oplus \mathcal{L}(\mathbf{e}_0, \mathbf{e}_1)$
7 $K \leftarrow \mathcal{K}(\mathbf{m}, (\mathbf{c}_0, \mathbf{c}_1))$
8 **return** $(K, (\mathbf{c}_0, \mathbf{c}_1))$
9 **end procedure**

(c) Decapsulation.

1 **procedure** `BIKE.Decaps`($\mathbf{sk}, (\mathbf{c}_0, \mathbf{c}_1)$)
2 $(\mathbf{h}_0, \mathbf{h}_1, \mu, \sigma) \leftarrow \mathbf{sk}$
3 $\mathbf{z} \leftarrow \mathbf{c}_0 \cdot \mathbf{h}_0$
4 $\mathbf{e}'_0, \mathbf{e}'_1 \leftarrow \texttt{Decode}(\mathbf{z}, \mathbf{h}_0, \mathbf{h}_1)$
5 $\mathbf{m}' \leftarrow \mathbf{c}_1 \oplus \mathcal{L}\left(\mathbf{e}'_0, \mathbf{e}'_1\right)$
6 **if** $\left(\mathbf{e}'_0, \mathbf{e}'_1\right) = \mathcal{H}(\mathbf{m}', \mu)$ **then**
7 $K \leftarrow \mathcal{K}(\mathbf{m}', (\mathbf{c}_0, \mathbf{c}_1))$
8 **else**
9 $K \leftarrow \mathcal{K}(\sigma, (\mathbf{c}_0, \mathbf{c}_1))$
10 **return** K
11 **end procedure**

Listing 1. BIKE KEM algorithms [ABB+24].

then mainly composed by the pair of sparse vectors $(\mathbf{h}_0, \mathbf{h}_1)$, and the public key is the dense vector $\mathbf{h}_{\text{pub}}$. Note that the matrices $[\text{rot}(\mathbf{h}_0) \,|\, \text{rot}(\mathbf{h}_1)] \in \mathbb{Z}_2^{r \times 2r}$ and $[\,\mathbf{I} \,|\, \text{rot}(\mathbf{h}_{\text{pub}})] \in \mathbb{Z}_2^{r \times 2r}$ are both parity check matrices of the same quasi-cyclic moderate-density parity-check (QC-MDPC) code. However, the first representation, only known by the secret key holder, allows for efficient syndrome decoding using bit-flipping algorithms.

The ciphertext resulting from the encapsulation is composed by two parts: $\mathbf{c}_0$ is a syndrome of $(\mathbf{e}_0, \mathbf{e}_1)$ with respect to a quasi-cyclic code, and $\mathbf{c}_1$ is an encryption of the seed $\mathbf{m}$. Note that this construction follows the FO transformation, ensuring that the receiver can recover the seed $\mathbf{m}$ and perform the reencryption to check if the received ciphertext is valid. Given the ciphertext $(\mathbf{c}_0, \mathbf{c}_1)$, the receiver first computes the secret syndrome $\mathbf{z} = \mathbf{c}_0 \cdot \mathbf{h}_0$. Notice that $\mathbf{z} = \mathbf{c}_0 \cdot \mathbf{h}_0 = (\mathbf{e}_0 + \mathbf{e}_1 \cdot \mathbf{h}_{\text{pub}}) \cdot \mathbf{h}_0 = \mathbf{e}_0 \cdot \mathbf{h}_0 + \mathbf{e}_1 \cdot \mathbf{h}_1$, so $\mathbf{z}$ is a syndrome of a low-weight vector with respect to a QC-MDPC code. Therefore, the receiver, who knows the secret sparse representation of the quasi-cyclic parity check matrix, can use QC-MDPC bit-flipping decoding algorithms to recover the sparse vector $[\mathbf{e}_0|\mathbf{e}_1]$ and compute the shared key K.

2.1.1 Decoding QC-MDPC codes in BIKE

The main step in BIKE decapsulation is the syndrome decoding with respect to a QC-MDPC code. BIKE's specification states that applications are free to choose a decoder, which may be tuned and optimized for particular platforms with specific constraints, provided that they achieve a negligible DFR [ABB+24, Section 4.2.1]. Until the end of Round 4, BIKE recommended a well-studied and implemented BGF decoder, but after Round 4, the new BIKE-Flip decoder [ABB+24] was proposed.

A limitation of BIKE-Flip is that it does not come with a constant-time implementation: the authors provide a reference implementation that uses floating-point arithmetic and secret-dependent branches. In the absence of a constant-time implementation, timing attacks are a natural adversary, and fault injection targeting a specific implementation becomes harder to justify, as it would not represent a fundamental security analysis of the scheme. In this work, we are primarily concerned with the official constant-time BGF decoder [DGK] referenced in the BIKE latest v5.2 specification [ABB+24] but we also show a generic attack against the BIKE-Flip decoder.

Given a syndrome ciphertext $\mathbf{c} = \mathbf{e}_0 + \mathbf{e}_1 \mathbf{H}_{\text{Pub}}^\top$, BGF [DGK20] is used to recover the sparse error vector $\mathbf{e} = [\mathbf{e}_0|\mathbf{e}_1]$ using the secret sparse matrix $\mathbf{H}$. The algorithm first computes the secret syndrome $\mathbf{z} = \mathbf{c}\mathbf{H}_0^\top$, then starts with $\mathbf{e} \leftarrow \mathbf{0}$ and performs a sequence of N_{Iter} iterations, each of which updates its knowledge of $\mathbf{e}$ until either $\mathbf{z} = \mathbf{e}\mathbf{H}^\top$ or the number of iterations exceeds a certain limit N_{Iter} and a decoding failure occurs.

Algorithm 1 presents the BGF decoder with its two bit-flipping auxiliary procedures:

Algorithm 1. The BGF decoding algorithm.

procedure $\texttt{BitFlipIter}(\mathbf{H}, \mathbf{z}, \mathbf{e}, \mathbf{s}, \tau_0)$
  $\texttt{BlackMask} \leftarrow \mathbf{0} \in \mathbb{F}_2^{2r}$
  $\texttt{GrayMask} \leftarrow \mathbf{0} \in \mathbb{F}_2^{2r}$
  for $j = 1$ to $2r$ **do**
    $\texttt{upc}_j \leftarrow \left|\mathrm{supp}\left(\mathbf{s}\right) \cap \mathrm{supp}\left(\mathbf{H}_j^\top\right)\right|$
    if $\texttt{upc}_j \geq \tau_0$ **then**
      $\mathbf{e}_j \leftarrow \overline{\mathbf{e}_j}$ ▷ Flips coordinate j of $\mathbf{e}$
      $\texttt{BlackMask}_j \leftarrow 1$
    else if $\texttt{upc}_j \geq \tau_0 - \delta$ **then**
      $\texttt{GrayMask}_j \leftarrow 1$
  $\mathbf{s} \leftarrow \mathbf{z} + \mathbf{e}\mathbf{H}^\top$ ▷ Recomputes the partial syndrome
  return $\mathbf{e}$, $\mathbf{s}$, $\texttt{BlackMask}$, $\texttt{GrayMask}$
end procedure
procedure $\texttt{BitFlipMaskedIter}(\mathbf{H}, \mathbf{z}, \mathbf{e}, \mathbf{s}, \texttt{Mask}, \tau_1)$
  for $j = 1$ to $2r$ **do**
    $\texttt{upc}_j \leftarrow \left|\mathrm{supp}\left(\mathbf{s}\right) \cap \mathrm{supp}\left(\mathbf{H}_j^\top\right)\right|$
    if $\texttt{Mask}_j = 1$ and $\texttt{upc}_j \geq \tau_1$ **then**
      $\mathbf{e}_j \leftarrow \overline{\mathbf{e}_j}$ ▷ Flips coordinate j of $\mathbf{e}$
  $\mathbf{s} \leftarrow \mathbf{z} + \mathbf{e}\mathbf{H}^\top$ ▷ Recomputes the partial syndrome
  return $\mathbf{e}$, $\mathbf{s}$
end procedure
procedure $\texttt{BGF}\left(\mathbf{H} = [\mathbf{H}_0|\mathbf{H}_1], \mathbf{z} = \mathbf{c}\mathbf{H}_0^\top\right)$
  $\mathbf{e} \leftarrow \mathbf{0} \in \mathbb{F}_2^{2r}$ ▷ Initializes the partial error vector
  $\mathbf{s} \leftarrow \mathbf{z}$ ▷ Initializes the partial syndrome
  for $i = 1$ to N_{Iter} **do**
    ▷ Every time $\mathbf{e}$ and $\mathbf{s}$ are updated, it holds that $\mathbf{s} = \mathbf{z} + \mathbf{e}\mathbf{H}^\top$
    if $i = 1$ **then**
      $\mathbf{e}, \mathbf{s}, \texttt{BlackMask}, \texttt{GrayMask} \leftarrow \texttt{BitFlipIter}\left(\mathbf{H}, \mathbf{z}, \mathbf{e}, \mathbf{s}, \tau_0 = \texttt{Thresh}\left(\mathbf{s}\right)\right)$
      $\mathbf{e}, \mathbf{s} \leftarrow \texttt{BitFlipMaskedIter}\left(\mathbf{H}, \mathbf{z}, \mathbf{e}, \mathbf{s}, \texttt{BlackMask}, \tau_1 = (d+1)/2 + 1\right)$
      $\mathbf{e}, \mathbf{s} \leftarrow \texttt{BitFlipMaskedIter}\left(\mathbf{H}, \mathbf{z}, \mathbf{e}, \mathbf{s}, \texttt{GrayMask}, \tau_1 = (d+1)/2 + 1\right)$
    else
      $\mathbf{e}, \mathbf{s} \leftarrow \texttt{BitFlipIter}\left(\mathbf{H}, \mathbf{z}, \mathbf{e}, \mathbf{s}, \tau_0 = \texttt{Thresh}\left(\mathbf{s}\right)\right)$ ▷ Ignores the black-gray masks
  if $\mathbf{e}\mathbf{H}^\top = \mathbf{z}$ **then** ▷ This condition is equivalent to $\mathbf{s} = \mathbf{0}$
    return $\mathbf{e}$
  else return $\perp$ ▷ Decoding failure
end procedure

`BitFlipIter` and `BitFlipMaskedIter`. These procedures are very similar to other iterative decoders, such as the original Gallager's bit-flipping algorithm [Gal62]. The value of δ is fixed as $\delta = 3$ for all security levels. While the number of iterations is fixed as $N_{\mathrm{Iter}} = 5$ for all security parameters, the threshold function `Thresh` varies, as defined in Table 2. We remark that the iterations used by BIKE-Flip are very similar to the `BitFlipIter` procedure, if we ignore the black and gray masks.

2.2 HQC

The algorithms used by HQC [GAMA+25] are described in Listing 2. Given the security level, the set of HQC parameters $(n_1, n_2, n, k, w, w_{\mathbf{r}}, w_{\mathbf{e}})$ is specified according to Table 3. Parameters (k_1, n_1) and (k_2, n_2) denote the dimension and length of the shortened Reed-Solomon (RS) code and the repeated Reed-Muller (RM) code, respectively. The concatenation of these two codes results in the powerful RMRS error correction code that encodes $8k_1$ bits into $n_1 n_2 \leq n$ bits. HQC operates on n-bit binary vectors, which are identified with polynomials in $\mathbb{Z}_2[x]/(x^n - 1)$, where n is taken as the smallest prime greater than $n_1 n_2$. The values w, $w_{\mathbf{r}}$, and $w_{\mathbf{e}}$ denote the weights of the sparse vectors used in key generation and encapsulation encryption.

When given an honestly generated ciphertext $\mathbf{c}$, the receiver uses the secret key $\mathbf{y}$ to

Table 3. HQC parameters for each security level [GAMA+25].

NIST Security	n	k	w	$w_{\mathbf{r}} = w_{\mathbf{e}}$	RS$[n_1, k_1, d_1]$	RM$[n_2, k_2, d_2]$	DFR
Level 1	17669	128	66	75	$[46, 16, 15]$	$[384, 8, 192]$	2^{-128}
Level 3	35851	192	100	114	$[56, 24, 16]$	$[640, 8, 320]$	2^{-192}
Level 5	57637	256	131	149	$[90, 32, 29]$	$[640, 8, 320]$	2^{-256}

(a) HQC internal PKE algorithms.

procedure HQC.PKE.KeyGen(No input)
  $\mathbf{x} \leftarrow\$ \left\{\mathbf{v} \in \mathbb{Z}_2^n : |\mathbf{v}| = w\right\}$
  $\mathbf{y} \leftarrow\$ \left\{\mathbf{v} \in \mathbb{Z}_2^n : |\mathbf{v}| = w\right\}$
  $\mathbf{h} \leftarrow\$ \mathbb{Z}_2^n$
  $\mathbf{s} \leftarrow \mathbf{x} + \mathbf{y} \cdot \mathbf{h}$
  $\mathbf{pk} \leftarrow (\mathbf{h}, \mathbf{s})$
  $\mathbf{sk} \leftarrow \mathbf{y}$
  return $(\mathbf{pk}, \mathbf{sk})$
end procedure

procedure HQC.PKE.Encrypt$(\mathbf{pk}, \mathbf{m}, \theta)$
  $\mathbf{h}, \mathbf{s} \leftarrow \mathbf{pk}$
  xof $\leftarrow$ Initialize XOF with seed θ
  $\mathbf{r}_2 \xleftarrow{\text{xof}} \left\{\mathbf{v} \in \mathbb{Z}_2^n : |\mathbf{v}| = w_{\mathbf{r}}\right\}$
  $\mathbf{e} \xleftarrow{\text{xof}} \left\{\mathbf{v} \in \mathbb{Z}_2^n : |\mathbf{v}| = w_{\mathbf{e}}\right\}$
  $\mathbf{r}_1 \xleftarrow{\text{xof}} \left\{\mathbf{v} \in \mathbb{Z}_2^n : |\mathbf{v}| = w_{\mathbf{r}}\right\}$
  $\mathbf{u} \leftarrow \mathbf{r}_1 + \mathbf{h} \cdot \mathbf{r}_2$
  $\mathbf{v} \leftarrow$ RMRS.Encode $(\mathbf{m}) + \mathbf{s} \cdot \mathbf{r}_2 + \mathbf{e}$
  return $(\mathbf{u}, \mathbf{v})$
end procedure

procedure HQC.PKE.Decrypt$(\mathbf{sk}, (\mathbf{u}, \mathbf{v}))$
  $\mathbf{c}' \leftarrow \mathbf{v} - \mathbf{u} \cdot \mathbf{y}$
  $\mathbf{m}' \leftarrow$ RMRS.Decode $(\mathbf{c}')$
  return $\mathbf{m}'$
end procedure

(b) HQC KEM algorithms.

procedure HQC.KEM.KeyGen
  $\mathbf{pk}, \mathbf{sk}' \leftarrow$ HQC.PKE.KeyGen (No input)
  $\sigma \leftarrow$ random 256-bit string
  $\mathbf{sk} \leftarrow \left(\mathbf{sk}', \mathbf{pk}, \sigma\right)$
  return $(\mathbf{pk}, \mathbf{sk})$
end procedure

procedure HQC.KEM.Encaps$(\mathbf{pk})$
  $\mathbf{m} \leftarrow$ random 256-bit string
  salt $\leftarrow$ random 256-bit string
  $K, \theta \leftarrow \mathcal{G}\left(\mathcal{H}\left(\mathbf{pk}, \mathbf{m}, \text{salt}\right)\right)$
  $(\mathbf{u}, \mathbf{v}) \leftarrow$ HQC.PKE.Encrypt $(\mathbf{pk}, \mathbf{m}, \theta)$
  $\mathbf{c} \leftarrow \left((\mathbf{u}, \mathbf{v}), \text{salt}\right)$
  return $(K, \mathbf{c})$
end procedure

procedure HQC.KEM.Decaps$(\mathbf{sk}, \mathbf{c})$
  $\mathbf{sk}', \mathbf{pk}, \sigma \leftarrow \mathbf{sk}$
  $(\mathbf{u}, \mathbf{v}), \text{salt} \leftarrow \mathbf{c}$
  $\mathbf{m}' \leftarrow$ HQC.PKE.Decrypt $\left(\mathbf{sk}', \mathbf{c}\right)$
  $K', \theta' \leftarrow \mathcal{G}\left(\mathcal{H}\left(\mathbf{pk}, \mathbf{m}', \text{salt}\right)\right)$
  $(\mathbf{u}', \mathbf{v}') \leftarrow$ HQC.PKE.Encrypt $(\mathbf{pk}, \mathbf{m}', \theta')$
  if $(\mathbf{u}', \mathbf{v}') = (\mathbf{u}, \mathbf{v})$ **then**
    return K'
  else
    ▷ Use implicit rejection key
    return $K_{\text{rej}} \leftarrow \mathcal{J}\left(\mathcal{H}\left(\mathbf{pk}, \sigma, \mathbf{c}\right)\right)$
end procedure

Listing 2. HQC algorithms.

compute $\mathbf{c}' = \mathbf{v} - \mathbf{y} \cdot \mathbf{u}$, which yields $\mathbf{c}' = \texttt{RMRS.Encode}(m) + \mathbf{x} \cdot \mathbf{r}_2 - \mathbf{y} \cdot \mathbf{r}_1 + \mathbf{e}$. Then, since the error $\mathbf{e}' = \mathbf{x} \cdot \mathbf{r}_2 - \mathbf{y} \cdot \mathbf{r}_1 + \mathbf{e}$ has a low weight, the receiver uses RMRS to remove the error $\mathbf{e}'$ by computing $\texttt{RMRS.Decode}\left(\mathbf{c}'\right)$ and hopefully recovers m. Note that, to achieve negligible DFR, the RMRS is designed to be very powerful.

3 Fault injection attack on HQC's polynomial multiplication

This section presents a key recovery attack against HQC based on skipping faults applied to the schoolbook polynomial multiplication. We begin the section by first stating the threat model and then describing the details of the fault location and its exploitation.

Attacker model. The attacker is given an HQC reference implementation compiled for Armv7-M considering `gcc 15.2.0` with flags `-mcpu=cortex-m7 -march=armv7e-m -O3`. In this paper, we always assume the FO-transform verification is heavily protected, so the attacker is not allowed to skip checks of the FO reencryption operation. We assume the attacker can apply one skipping fault on each decapsulation procedure, but they can

Algorithm 2. The Karatsuba multiplication algorithm.

```
1  procedure Karatsuba(a, b, n)
2      if n ≤ τ then                ▷ Precomputed threshold for which schoolbook is efficient when n ≤ τ
3          return SchoolbookMultiplication(a, b)
4      (a_0, a_1) ← Two halves of a                ▷ Split a into parts of lengths ⌊n/2⌋ and ⌈n/2⌉
5      (b_0, b_1) ← Two halves of b                ▷ Split b into parts of lengths ⌊n/2⌋ and ⌈n/2⌉
6      r_0 ← Karatsuba(a_0, b_0, ⌊n/2⌋)
7      r_1 ← Karatsuba(a_1, b_1, ⌈n/2⌉)
8      r_2 ← Karatsuba(a_0 + a_1, b_0 + b_1, ⌈n/2⌉)
9      r ← [r_0 | (r_0 + r_1 + r_2) | r_1]          ▷ Reconstruct result concatenating partials
10     return r
11 end procedure
```

change the fault location each time. After faulting the decapsulation, the only result that is observable to the attacker is whether the decapsulation failed or succeeded. We remark that the ability to apply faults to different locations is a theoretical assumption. While we are able to demonstrate skips to a single point in practice (see Section 6), the synchronization requirements to apply faults to different precise locations would require a more powerful attacker.

3.1 Fault location

To understand the fault attack vulnerability in HQC, we first examine how fault injection affects the schoolbook polynomial multiplication employed within the scheme. The decryption algorithm in HQC starts by taking the ciphertext $(\mathbf{u}, \mathbf{v})$ and computing $\mathbf{c}' = \mathbf{v} - \mathbf{y} \cdot \mathbf{u}$, using the secret key $\mathbf{y}$. To compute the multiplication $\mathbf{y} \cdot \mathbf{u}$, the Karatsuba and Toom-Cook algorithms are used by the HQC implementation. These are well-known algorithms that recursively split the polynomial into equally sized parts, and combine the results of recursive calls to build partial products. In the base case, a schoolbook multiplication algorithm is used to multiply two small polynomials.

While different combinations of Toom-Cook and Karatsuba are used in the optimized AVX2 implementation, the reference implementation uses Karatsuba directly. Therefore, we focus on the Karatsuba algorithm, which is shown in Algorithm 2. However, our arguments should also apply to Toom-Cook multiplication, which can be seen as a generalization of the Karatsuba algorithm.

Let us consider what happens when Algorithm 2 is applied to $\mathbf{y}$ and $\mathbf{u}$. We can see that the recursive calls defining $\mathbf{r}_0$ and $\mathbf{r}_1$ will directly process the two parts $\mathbf{y}_0$ and $\mathbf{y}_1$ of the secret $\mathbf{y}$. By recursively applying this thought, we end up noticing that the schoolbook multiplication algorithm will run on every chunk of τ bits of the secret $\mathbf{y}$.

Figure 1 illustrates, with a toy example of a 32-bit secret, how two levels of Karatsuba calls end up having schoolbook multiplications processing the four 8-bit chunks. This makes the schoolbook multiplication an interesting point to apply a failure: it processes the key and has a somewhat cascading effect in the final product.

The implementation of the schoolbook multiplication is shown in Listing 3. Let us consider what happens when it is processing a chunk of the secret key $\mathbf{y}$ passed as argument `a`. The algorithm iterates over each bit of `a` and updates the considered product by adding the corresponding circular shift of `b`. Following constant-time programming practices, the code uses a mask variable so that it does not branch depending on the secret bits of `a`.

We know that each bit of the secret key $\mathbf{y}$ will eventually be processed by the function `schoolbook_mul`. Suppose the implementation uses 64-bit registers and that an attacker is able to skip the AND instruction in Line 6. Let us evaluate the result of this fault in the variable `mask`, considering the case when `a[i]` is a 64-bit chunk of $\mathbf{y}$. There are three possible cases for `a[i]`: it can have only null bits, exactly one non-null bit, and more than

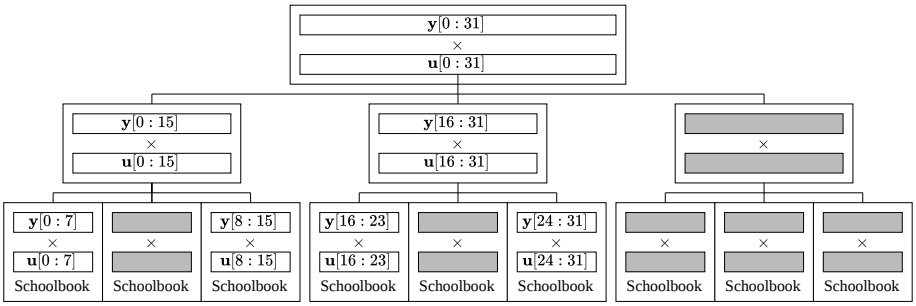


Figure 1. Toy example showing how, when computing $\mathbf{y} \cdot \mathbf{u}$ using 2 levels of Karatsuba, the four 8-bit parts of the 32-bit secret $\mathbf{y}$ are processed by calls to the schoolbook algorithm.

```
1  static void schoolbook_mul(uint64_t *r, uint64_t *a, uint64_t *b, size_t n) {
2    memset(r, 0, 2 * n * sizeof(uint64_t));
3    for (size_t i = 0; i < n; i++)
4      for (int bit = 0; bit < 64; bit++) {
5        // Remember that, in Armv7-M, mask will be stored in a 32-bit register
6        uint64_t mask = -((a[i] >> bit) & 1);  // Fault location A (and instruction in -O3)
7        int inv = 64 - bit;
8        if (bit == 0)                          // Fault location B (cbnz instruction in -Os)
9          for (size_t j = 0; j < n; j++)
10           r[i + j] ^= b[j] & mask;
11       else
12         for (size_t j = 0; j < n; j++) {
13           r[i + j]     ^= (b[j] << bit) & mask;
14           r[i + j + 1] ^= (b[j] >> inv) & mask;
15         }
16     }
17 }
```

Listing 3. Reference C implementation of the schoolbook polynomial multiplication.

one non-null bit. When it has only null bits, i.e. `a[i]` $= 0$, then the skip has no effect since `mask` $= 0$ is exactly the value it should have been without the fault.

Now suppose `a[i]` has exactly one non-null bit in position k. Then

$$\texttt{(a[i] >> bit)} = \begin{cases} \texttt{1 << (k - bit)}, & \text{if } \texttt{bit} < k, \\ 1, & \text{if } \texttt{bit} = k, \\ 0, & \text{if } \texttt{bit} > k. \end{cases}$$

Therefore, based on the two's-complement representation in 64-bit registers, the faulted mask value can be defined as

$$\texttt{mask} = \texttt{-(a[i] >> bit)} = \begin{cases} \texttt{0b}\underbrace{\texttt{11111...11111}}_{64-(k-\texttt{bit})}\underbrace{\texttt{00000...00000}}_{k-\texttt{bit}}, & \text{if } \texttt{bit} < k, \\ \texttt{0b11111...11111}, & \text{if } \texttt{bit} = k, \\ \texttt{0b00000...00000}, & \text{if } \texttt{bit} > k. \end{cases} \tag{1}$$

This means that, when $\texttt{bit} \geq k$, the value of `mask` is exactly the same as it would have been if the fault was not applied. However, when $\texttt{bit} < k$ we get a different `mask` value, which then can be catastrophic. In particular, the loops in Lines 9 and 12 using the faulted mask will result in an update of n entries of the array `r` with an XOR with garbage bits (e.g. `(b[j] << bit) & mask`), and may result in a decryption failure.

We still have to consider the third case, when `a[i]` has $\eta > 1$ non-null bits, which gets exponentially more rare when η increases, since $\mathbf{y}$ is sparse. While the analysis is more convoluted than in the $\eta \leq 1$ case, it is possible to observe a similar behavior for $\eta > 1$.

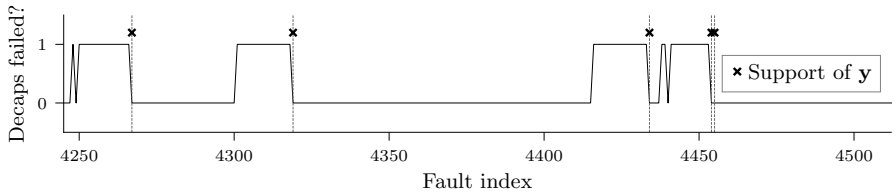


Figure 2. The output of the data collection phase when attacking HQC-128 using one fault per decapsulation for a selected index range. The skipping faults are applied to the schoolbook multiplication when processing different indexes of $\mathbf{y}$.

In particular, if $k_1 < \ldots < k_m$ are the positions of the non-null bits of `a[i]`, we observe that: (*i*) `mask` $\approx$ `0b111...111`[1] when `bit` $= k_i$, (*ii*) `mask` $= 0$ when `bit` $> k_m$ and (*iii*) `mask` $\neq 0$ otherwise. These are the main observations allowing us to recover the key, as explained next.

3.2 Key recovery attack

Our attack strategy works in two phases: data collection, when the fault is applied and the decapsulation results are observed, which is then followed by key recovery.

Step 1: Data collection. This step runs faulted decapsulations of valid ciphertexts and records whether they were successful or not. For each $k = 0$ to $n - 1$, we skip the `AND` operation corresponding to the processing of bit $\mathbf{y}[k]$ of the secret key inside a call to the schoolbook multiplication procedure.[2] We let $f_k \in \{0, 1\}$ be the output of the kth run, by setting $f_k = 1$ or $f_k = 0$ when the decapsulation failed or succeeded, respectively.

Figure 2 illustrates the output of this first step. We can see that the positions in the support of $\mathbf{y}$ typically lie in points where we move from a decapsulation failure to a success. Now, it is important to note that the reverse is not true: there are points where we move from a failure to a success, which are not in the support. This behavior is a direct consequence of Equation 1: when we see a sequence of faults (case `bit` $< k$), followed by a successful decapsulation (case `bit` $= k$), then $\mathbf{y}[k]$ is a potential non-null bit.

Step 2: Key recovery. Consider the results $f_0, \ldots, f_{n-1}$ obtained from the data collection step. Let us consider the limitations of the information that can be learned from $f_0, \ldots, f_{n-1}$. Since Equation 1 is defined for a 64-bit block of $\mathbf{y}$, flips from a failure in f_{k-1} to a success in f_k only gives useful information when bits $\mathbf{y}[k-1]$ and $\mathbf{y}[k]$ fall within the same 64-bit block. This implies that there is no way to use the failures to learn the first bit of each 64-bit block (i.e. bits $\mathbf{y}[64i]$, for $0 \leq i < \lceil n/64 \rceil$). The procedure is also not able to correctly detect an element k in the support of $\mathbf{y}$ if $k - 1$ is also in the secret support, as the drop in the plot is only observed for the first element. Therefore, fault injection alone is not able to provide the full secret key, and we need additional steps to obtain the remaining bits of the key.

Our strategy begins with Algorithm 3, which uses the output $f_0, \ldots, f_{n-1}$ from the data collection phase to build a set Z of coordinates of $\mathbf{y}$ that we can assume to be zero. Starting with $Z = \{0, 1, \ldots, n-1\}$, we can see that Line 4 removes indexes that are multiples of 64 from Z, since we cannot get information on them, as discussed above.

[1] We write `mask` $\approx$ `0b111...111` because there may be a few null bits in `mask` that are caused by the negation of the multiple non-null bits in `a[i]`.

[2] The processing of bit $\mathbf{y}[k]$ by the `schoolbook_mul` function is done by a specific call to `schoolbook_mul` (see Figure 1) and a particular pair of indices (`i`, `bit`) in Listing 3.

Algorithm 3. Extracting set Z of known null entries of $\mathbf{y}$ from faults.

```
1  procedure BuildZ(Decapsulation results f_0, ..., f_{n-1} from data collection phase)
2      Z ← {0, ..., n − 1}                          ▷ Start with the full set of indexes
3      for each i = 0 to ⌈n/64⌉ − 1 do              ▷ Iterate over the 64-bit blocks
4          Z ← Z − {64i}                            ▷ Remove multiples of 64 as faults give no information
5          for each j = 1 to 63 do                  ▷ Iterate over the bits of the block, skipping first
6              k ← 64i + j
7              if k ≥ n then break                  ▷ Break when finished processing the last block
8              if f_k = 0 and f_{k−1} = 1 then      ▷ Condition to identify non-null entries
9                  Z ← Z − {k}                      ▷ Remove the non-null entry found
10                 for ℓ = 1 to γ − 1 do            ▷ γ depends on HQC parameters (n, w)
11                     if j + ℓ ≥ 64 then break     ▷ Break when overflowing to next 64-bit block
12                     Z ← Z − {k + ℓ}              ▷ Remove potential non-null neighbors of k
13     return Z
14 end procedure
```

Then, Line 9 removes indexes for which a decapsulation failure moved to a success, which are associated with non-null entries. However, remember that we cannot reliably detect a series of non-null entries in $\mathbf{y}$ from the sequence $f_0, \ldots, f_{n-1}$ alone. Therefore, we use a parameter γ to conservatively avoid the undetectable cases when there are sequences of γ non-null entries in the secret key. This is done by Line 12, that, once a potential non-null entry k is found, removes the values $k+1, \ldots, k+\gamma-1$ from Z.

A good value for γ ensures that our attack is able to recover the secret key with high probability. In other words, γ should be enough to ensure a negligible probability of a random secret vector $\mathbf{y}$ containing a sequence of γ non-null bits. For HQC parameters (n, w), we can bound the probability that a random n-bit binary $\mathbf{y}$ of weight w has a sequence of γ non-null bits as follows. Since $\Pr(\mathbf{y}[k] = 1) = w/n$, the probability that $\mathbf{y}[k] = \ldots = \mathbf{y}[k+\gamma-1] = 1$ for a fixed index k is approximately $(w/n)^\gamma$. Then, applying the union bound on the n possible values of k, we get

$$\Pr(\text{Attack fails}) = \Pr(\mathbf{y}[k] = \mathbf{y}[k+1] = \ldots = \mathbf{y}[k+\gamma-1] = 1 \text{ for any } k) \approx n\left(\frac{w}{n}\right)^\gamma.$$

Now with Z in hand, we can proceed to solve the key equation $\mathbf{x} + \mathbf{y} \cdot \mathbf{h} = \mathbf{s}$ for secret $(\mathbf{x}, \mathbf{y})$ and public key $(\mathbf{h}, \mathbf{s})$. We can employ well-known techniques, such as information set decoding (ISD) algorithms assisted by knowledge on Z. Namely, first write the key equation in matrix form as $\begin{bmatrix}\mathbf{I} & \text{rot}(\mathbf{h})\end{bmatrix}\begin{bmatrix}\mathbf{x} & \mathbf{y}\end{bmatrix}^\top = \mathbf{s}$. We know $|Z|$ zero coordinates of $\mathbf{y}$, making this a linear system with n equations on $2n - |Z|$ unknowns. Thus, if we correctly guess $n - |Z|$ zero coordinates of $\mathbf{x}$, we could solve the key equation, and if not, we can keep trying until we guess them correctly. Since $\mathbf{x}$ has $n - w$ zero coordinates in total, the probability of correctly guessing $n - |Z|$ zero coordinates of $\mathbf{x}$ is

$$\Pr(\text{Guess } n - |Z| \text{ non-null entries of } \mathbf{x}) = \binom{n-w}{n-|Z|} \Big/ \binom{n}{n-|Z|}.$$

Simulation results for HQC-128. To demonstrate the feasibility of this attack for key recovery, we performed a simulation considering 50 HQC-128 keys. The attack uses $n = 17669$ faults related to indexes k that are not multiple of 64, each applied to a single decapsulation. Note that HQC-128 uses parameters $(n, w) = (17669, 66)$, thus the attack parameter $\gamma = 4$ is enough for the attack to fail only for a fraction 2^{-18} of the keys. After processing the output of the data collection phase to build the set Z, we observed a median size of $|Z| = 17072$. The average cost of guessing $n - |Z|$ zero coordinates of $\mathbf{x}$ to finish the key equation was lower than 10 guesses, making the attack practical for an attacker that can implement this fault model.

We remark that it is possible to reduce the number of decapsulations from n to $n - \lceil n/64 \rceil = 17392$ by setting $f_{64i} = 1$, for $0 \leq i < \lceil n/64 \rceil$, instead of computing the real value experimentally. This is conservative, since it results in strictly more indexes being removed from Z. While it increases the number of guesses to 600, this is still fairly small.

Limitations of the attack. Although this attack only needs one fault per decapsulation to skip the same instruction, the skip is done at different points in time. Specifically, for each target index k of the the secret key $\mathbf{y}$, we skip the `AND` instruction of a specific loop iteration of one particular call to the schoolbook multiplication algorithm. This may be harder to mount than a simple skipping fault that is applied to exactly the same point in time, potentially limiting its applicability in a realistic scenario.

Moreover, due to the shared similarities between HQC and BIKE, one might ask if a similar attack could be mounted against BIKE. In particular, BIKE also uses the Karatsuba algorithm to compute $\mathbf{c} \cdot \mathbf{h}_0$, for a ciphertext $\mathbf{c}$ and sparse secret $\mathbf{h}_0$. We investigated this possibility, and, although it seems possible to apply faults to the base multiplications processing parts of the secret, the propagation of the fault is very limited because the BGF decoder recomputes the syndrome on every iteration.

4 A general method for fault attacks on code KEMs

In this section, we present a general method that enables fault injection attacks on BIKE and HQC even when the FO reencryption check is heavily protected.

Fault injections beyond the Fujisaki-Okamoto reencryption check. Our main motivation is to provide a generic procedure to find and exploit faults on BIKE and HQC that are able to recover the key even against implementations that heavily protect the FO verification, unlike previous fault attacks [XIU+21]. In particular, we assume that the attacker can apply faults to points in the decapsulation process (e.g. the IND-CPA decryption or re-encryption), but not to the ciphertext equality check.

We follow the reaction attack model, in which the attacker can determine whether the decapsulation succeeded or not. We remark that the victim's reaction may give information on the success or failure of the decapsulation even under the implicit rejection variant of the FO transformation. For concreteness, consider the real-world use of KEMs: two parties use a KEM to share a symmetric key that is used within a larger protocol or for direct communication using AEAD schemes. Then, if the attacker encapsulates a key and the receiver does not derive the same key due to implicit rejection, then their communication will fail at some point (e.g. due to an invalid authentication tag), which the attacker interprets as a decapsulation failure. Current versions of BIKE and HQC are not vulnerable to reaction attacks due to their negligible DFR for valid ciphertexts. While invalid ciphertexts can increase the DFR, these always cause decapsulation failures because of the FO reencryption check, thus not providing useful information to the attacker. In what follows, we propose a new method that uses fault injection to increase the DFR even for *valid ciphertexts*, enabling reaction attacks capable of recovering the secret key.

4.1 Our method

Let us consider the similarities shared by HQC and BIKE. In both schemes the secret key is a pair of low-weight vectors: $(\mathbf{h}_0, \mathbf{h}_1)$ for BIKE, and $(\mathbf{x}, \mathbf{y})$ for HQC.

When BIKE and HQC decrypt ciphertexts $\mathbf{c}$ and $(\mathbf{u}, \mathbf{v})$, respectively, they compute:

$$\begin{cases} \texttt{BGF.Decode}\left(\mathbf{c} \cdot \mathbf{h}_0\right) = \texttt{BGF.Decode}\left(\mathbf{e}_0 \cdot \mathbf{h}_0 + \mathbf{e}_1 \cdot \mathbf{h}_1\right), \text{ and} \\ \texttt{RMRS.Decode}\left(\mathbf{v} - \mathbf{u} \cdot \mathbf{y}\right) = \texttt{RMRS.Decode}\left(\texttt{RMRS.Encode}\left(m\right) + \mathbf{x} \cdot \mathbf{r}_2 - \mathbf{y} \cdot \mathbf{r}_1 + \mathbf{e}\right). \end{cases}$$

We can see that the vectors to be decoded in both cases have a similar structure: $\mathbf{e}_{\text{BIKE}} = \mathbf{e}_0 \cdot \mathbf{h}_0 + \mathbf{e}_1 \cdot \mathbf{h}_1$ for BIKE, and $\mathbf{e}_{\text{HQC}} = \mathbf{x} \cdot \mathbf{r}_2 - \mathbf{y} \cdot \mathbf{r}_1$ for HQC. These consist of sums of products of sparse vectors, where some of them are known by the attacker (sender) and the others are parts of the receiver's secret key.

The relation between the error to be corrected and the secret keys. It is well-known that the Hamming weights of $\mathbf{e}_{\text{BIKE}}$ and $\mathbf{e}_{\text{HQC}}$ highly depend on an important relation between the secret key vectors and the encryption nonces known by the attacker. This relation was first observed by Guo et al. [GJS16] in 2016 in their famous reaction attack on QC-MDPC (the predecessor of BIKE). The same relation was later explored against HQC in a timing attack [PT20]. Guo et al. [GJS16] showed that the Hamming weight of $\mathbf{a} \cdot \mathbf{b}$ tends to be smaller when the spectrums of $\mathbf{a}$ and $\mathbf{b}$ share a high number of entries. This means that information on the weight of $\mathbf{e}_{\text{BIKE}}$ or $\mathbf{e}_{\text{HQC}}$ can be exploited by an attacker for key recovery. While the weights of these vectors are not directly accessible to attackers, errors of larger weight are typically harder to decode. Therefore, a decapsulation failure could leak if the weight of $\mathbf{e}_{\text{BIKE}}$ or $\mathbf{e}_{\text{HQC}}$ is high, enabling a reaction attack.

Using faults to obtain information on the Hamming weight of the error. Our idea is to use faults in points of the decapsulation in a way that it increases the Hamming weight of the error to be corrected by HQC or BIKE decoder ($\mathbf{e}_{\text{BIKE}}$ or $\mathbf{e}_{\text{HQC}}$) in a somewhat predictable manner. In particular, we want faults that add controlled errors to these vectors so that we start seeing some decapsulation failures. If we are successful, a critical behavior takes place: a decapsulation error or success will be correlated with the Hamming weight of $\mathbf{e}_{\text{BIKE}}$ or $\mathbf{e}_{\text{HQC}}$. This can then be traced back to relations with the spectrum of secret vectors. For example, a fault in the BIKE decapsulation procedure enables us to obtain $\mathbf{e}'_{\text{BIKE}} = \mathbf{e}_{\text{BIKE}} + \mathbf{e}'$, where $\mathbf{e}'$ is a small binary noise. Then $\texttt{BGF.Decode}\left(\mathbf{e}'_{\text{BIKE}}\right) = \texttt{BGF.Decode}\left(\mathbf{e}_{\text{BIKE}} + \mathbf{e}'\right)$.

So, if $\mathbf{e}'$ is small enough and we see a decapsulation failure, then we could infer that the Hamming weight of $\mathbf{e}_{\text{BIKE}}$ is large, which in turn gives us the information that the spectrums of $\mathbf{h}_0$ and $\mathbf{e}_0$ are related (as well as those from $\mathbf{h}_1$ and $\mathbf{e}_1$). Naturally, we have to be careful not to cause larger errors $\mathbf{e}'$, otherwise the decapsulation will always fail and thus will not give us any information.

Key recovery attack. The key recovery attack can then proceed exactly as done in spectrum-based attacks against BIKE and HQC [GJS16, PT20]. For concreteness, we explain the key recovery procedure for HQC, but the same idea is directly applicable to BIKE if we map HQC secrets $(\mathbf{x}, \mathbf{y})$ to BIKE secrets $(\mathbf{h}_0, \mathbf{h_1})$, and the HQC nonces $(\mathbf{r}_2, \mathbf{r_1})$ to BIKE nonces $(\mathbf{e}_0, \mathbf{e}_1)$.

The attacker proceeds by generating ciphertexts to the target while keeping track of the nonces $(\mathbf{r}_1, \mathbf{r_2})$. By checking whether the decapsulation failed or not, the attacker can build two arrays $\mathbf{t_x}$ and $\mathbf{t_y}$ such that $\mathbf{t_x}[d]$ (resp. $\mathbf{t_y}[d]$) is the average number of failures when the cyclic distance d appears in the spectrum of $\mathbf{x}$ (resp. $\mathbf{y}$). This procedure enables $\mathbf{t_y}[d]$ to act as a distinguisher of whether d appears in the spectrum of $\mathbf{y}$ as follows. Lower values of $\mathbf{t_y}[d]$ correlate with lower $|\mathbf{e}_{\text{HQC}}|$, which are also associated with lower $|\mathbf{y} \cdot \mathbf{r}_2|$, which is finally correlated with d not appearing in the spectrum of $\mathbf{y}$. A real-world histogram of a vector like $\mathbf{t_x}$ is shown in Figure 4b, which illustrates how values in $\mathbf{t_x}$ correspond to entries in the spectrum of $\mathbf{x}$.

Once values of $\mathbf{t_y}[d]$ for distances that are in and out of the spectrum of $\mathbf{y}$ are sufficiently separated, one can apply key reconstruction algorithms to recovery the key. Guo et al. [GJS16] proposed a depth-first search algorithm for reconstructing the key given the spectrum that is simple and effective, but requires the full set of distances in the spectrum of the secret key. Later, Paiva and Terada [PT20] proposed a randomized variant targeting the case when the attacker has partial information on the secret key spectrum.

4.2 An improved algorithm for key reconstruction from spectrum

We now present an overview of our new algorithm for key reconstruction from spectrum. The main motivation for introducing this procedure is empirical: in our experiments, previously proposed reconstruction algorithms [GJS16, PT20] were unable to fully recover the secret key in the fault-attack setting considered in Section 5.2.

Let $\mathbf{y}$ be the secret n-bit vector of Hamming weight w that we wish to reconstruct. Assume we are given two sets of distances, D_{in} and D_{out}, representing distances that are believed to be in and out of the spectrum of $\mathbf{y}$. In practice, these sets are obtained from the observed values $\mathbf{t}_{\mathbf{y}}[d]$: distances in D_{in} correspond to values that are among the smallest, while distances in D_{out} correspond to values that are among the largest ones. A standard reconstruction strategy then proceeds iteratively, adding support positions while trying to enforce consistency with these two sets.

Algorithm 4 shows our new algorithm, which builds up on the randomized search strategy of Paiva and Terada [PT20], which primarily uses D_{out} to exclude invalid candidates, but incorporates three additional heuristics: (*i*) conservative first iterations, (*ii*) scoring candidates by consistency with D_{out}, and (*iii*) completion by selecting unlikely null positions. The functions for scoring candidates is at the core of our heuristics, which is also parametrized by two parameters θ_1 and θ_2, which define the behavior in particular iterations, as explained next.

The first iterations are particularly critical, since an incorrect early choice can easily drive the search toward a wrong support. To improve the quality of the initialization, during the first θ_1 iterations we only accept a candidate position c if all distances induced between c and the currently guessed support belong to D_{in}. This makes the early support estimate more conservative, but also more reliable.

After the initial phase, the algorithm evaluates each remaining candidate through a score that measures how consistent it is with the set of already identified zero positions. More precisely, for a candidate c, we compute the set of new coordinates that would be forced to zero by D_{out}, and we score c according to the overlap of this set with the current set of indexes that are considered to be zeros. Candidates with larger overlap are preferred, since they are more compatible with the exclusions accumulated so far. Intuitively, a correct support position should reinforce the zero-pattern already suggested by D_{out}.

A single execution of the iterative typically is not able to recover all w support positions when D_{out} contains some incorrect distances that are in the spectrum of $\mathbf{y}$. To deal with this case, when the recovered support is already sufficiently large, we apply a completion step. For the current partial support, we compute a rating for the positions currently classified as null and select the positions with the smallest ratings, that is, those that appear least likely to be truly null according to D_{out}.

Comparison with other approaches. Although we developed these improvements targeting the specific application of this paper, our results suggest that this algorithm presents significant improvements over the previous state of the art [PT20]. In particular, our Python implementation was able to solve in 1 minute an instance [PT20, Table 2] that their C implementation was not able to solve even when given hours of runtime. It is important to note that recently Salom et al. [SSV26] presented a new algorithm that uses soft information for key reconstruction. While we did not run a comparison with their algorithm due to its recent appearance, their work seems very well suited to our setting.

5 Applying our general method to BIKE and HQC

In this section, we discuss how our general method can be used to find and exploit faults for full key recovery in BIKE and HQC. We show how simply skipping arithmetic and

Algorithm 4. Key reconstruction from spectrum information.

```
procedure GetScoresForCandidates(candidates, support, zeros, D_out, D_in, strict)
    scores ← Empty dictionary mapping integers to integers
    for c in candidates do
        if strict = True then
            if |support| ≤ θ_1 and dist(s, c) ∉ D_in for any s ∈ support then
                scores[c] = 0
                continue
        tmp_zeros ← {c + d : d ∈ D_out} ∪ {c − d mod n : d ∈ D_out}
        scores[c] = |tmp_zeros ∩ zeros|
    return scores
end procedure
procedure KeyReconstruction(D_in, D_out, n)
    s ← Sample from D_in
    support ← {0, s}
    zeros ← {d : d ∈ D_out} ∪ {−d mod n : d ∈ D_out}          ▷ Set of necessarily null entries
    candidates ← [n] − zeros − support
    while |support| < w do
        if |candidates| > 0 then
            scores ← GetScoresForCandidates(candidates, support, zeros, D_out, D_in, True)
            best_candidates ← η candidates with the highest scores
            r ← Random element from best_candidates
            support ← support ∪ {r}
            zeros ← zeros ∪ {r + d : d ∈ D_out} ∪ {r − d mod n : d ∈ D_out}
            candidates ← [n] − zeros − support
        else if |support| > θ_2 then
            candidates ← [n] − support                        ▷ Any new entry is now a candidate
            scores ← GetScoresForCandidates(candidates, support, zeros, D_out, D_in, False)
            S ← Set of the w − |support| keys in scores with the highest values
            support ← support ∪ S                             ▷ Support is now complete with w entries
    return support
end procedure
```

logical Armv7-M instructions such as `subs` or `ands` or skipping loops can result in attacks against both schemes. We remark that the objective of this section is not to perform an exhaustive evaluation of possible points, but we try to focus on two points, one for each scheme, that allow us to assess the generality of the method.

Attacker model. The following sections assume a similar model to the one considered in Section 3, but with a significant difference: now the attacker can only apply one skipping fault to a fixed point in time during the execution of the decapsulation. This means that the following attacks can be carried by less powerful attackers, as they do not need to change the fault location in each decapsulation. More specifically, the attacker is given reference implementations for HQC and BIKE compiled for Armv7-M considering `gcc 15.2.0` with flags `-mcpu=cortex-m7 -march=armv7e-m -O3`, and we assume the FO-transform verification is heavily protected. Following the reaction attack model, after faulting the decapsulation the attacker can learn whether the decapsulation failed or succeeded.

5.1 Fault attack on the BIKE decoding function

To apply our method to BIKE, we need a location for a skipping fault with a global effect during the decoding, since faults that have only localized effects end up having little impact on the decapsulation failure. A natural place to apply the fault is during the decoding step, which uses the secret key since it requires the sparse parity check matrix of the code. We mainly focus on the BGF decoder, which is the latest BIKE decoder to have a constant-time implementation without floating point operations, making it a more

```c
static inline uint64_t mul64high(uint64_t a, uint64_t b) {
  uint64_t a_lo, a_hi, b_lo, b_hi;
  a_lo = a & 0xffffffffULL;
  b_lo = b & 0xffffffffULL;
  a_hi = a >> 32;
  b_hi = b >> 32;
  return a_hi * b_hi + ((a_hi * b_lo + a_lo * b_hi) >> 32);
}

// Return 0 if v1 < v2, (-1) otherwise
static __inline__ uint32_t
secure_l32_mask(uint32_t v1, uint32_t v2) {
  return ~((uint32_t)(((uint64_t)v1 - (uint64_t)v2) >> 32));
}

uint8_t get_threshold(uint64_t syndrome_weight) {
  uint64_t thr = T0 + (T1 * syndrome_weight);
  thr = mul64high(thr, T_MUL_CONST);
  thr >>= T_SHIFT;
  uint32_t mask = secure_l32_mask((uint32_t) thr, T_MIN);
  thr = (mask & thr) | ((~mask) & T_MIN);
  return thr;
}
```

```asm
get_threshold:
    ldr     r2, .L3
    ldr     ip, .L3+4
    umull   r0, r3, r0, r2
    mla     r1, r2, r1, r3
    ldr     r3, .L3+8
    ldr     r2, .L3+12
    adds    r0, r0, r3
    adc     r1, r1, #0
    umull   r0, r3, r0, r2
    umlal   r0, r3, r1, ip
    movs    r0, #0
    umlal   r3, r0, r1, r2
    lsrs    r3, r3, #26
    orr     r3, r3, r0, lsl #6
    subs    r2, r3, #36 // Point A
    sbc     r0, r0, r0
    mvns    r2, r0
    and     r0, r0, #36
    ands    r2, r2, r3 // Point B
    orrs    r0, r0, r2
    uxtb    r0, r0
    bx      lr
```

Listing 4. Implementation of the BGF threshold computation in C and the corresponding Assembly code.

suitable target (see Section 2.1). However, we also briefly describe an attack on the newer BIKE-Flip decoder in the end of this section.

Fault location. We know that BGF's most important iterations are the first ones [PT22]. Thus, if we can successfully apply a fault with global effect in one of the first iterations, the failure may propagate through the whole decoding, and hopefully result in a failure under certain conditions (e.g., the secret syndrome has a slightly larger weight). One point that has a global effect on the iteration is the computation of the BGF threshold. This threshold, as shown in Algorithm 1, is used when deciding to flip each bit, and is fixed during the whole iteration. Therefore, if we are able to apply a fault to the threshold computation in an early iteration, we might be able to cause decryption failures.

Listing 4 shows how the computation of the BGF threshold is computed in BIKE's reference implementation. The implementation follows constant-time practices and avoids floating-point arithmetic by using scaled constants `T0` and `T1` for the computation of a linear function, which is then scaled back by shifting the value by `T_SHIFT`. Furthermore, a condition mask is used to compute the maximum between the linear function and a fixed value `T_MIN`. Namely, the variable `mask` is set to 0 when the computed threshold is lower than `T_MIN` and `0xFFFFFFFF` otherwise.

Consider what happens when a skipping fault is applied during the computation of the `and` instruction between `mask` and `thr`, resulting in `thr = mask | ((~mask) & T_MIN)`. For reference, this fault location corresponds to Lines 21 (C implementation) and 20 (Assembly) in Listing 4. Since the variable `mask` is a 32-bit mask equal to 0 when condition $\texttt{thr} < \texttt{T_MIN}$ is met, we have

$$\texttt{thr} = \begin{cases} \texttt{T_MIN}, & \text{if } \hat{\texttt{T0}} \cdot |\mathbf{s}| + \hat{\texttt{T1}} < \texttt{T_MIN}, \\ 2^{32} - 1, & \text{otherwise,} \end{cases}$$

for appropriate values $\hat{\texttt{T0}}$ and $\hat{\texttt{T1}}$, corresponding to scaled versions of `T0` and `T1`.

Applicability of our general attack method. So let us analyze the behavior of the `get_threshold` function when this fault is applied. Intuitively, `thr` is a valid threshold value (`T_MIN`) when $\hat{\texttt{T0}} \cdot |\mathbf{s}| + \hat{\texttt{T1}} < \texttt{T_MIN}$, but a very large value otherwise. In particular, this very large threshold would be too high for the iteration to flip any bit. This means

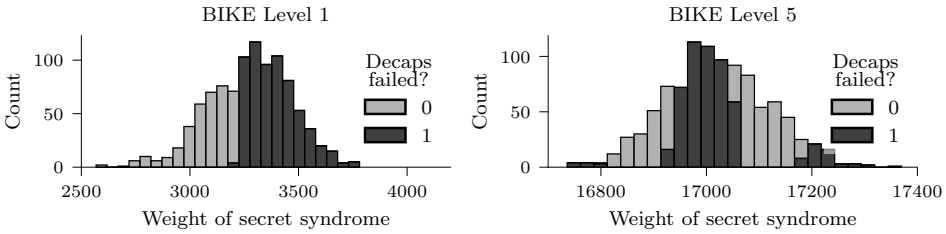
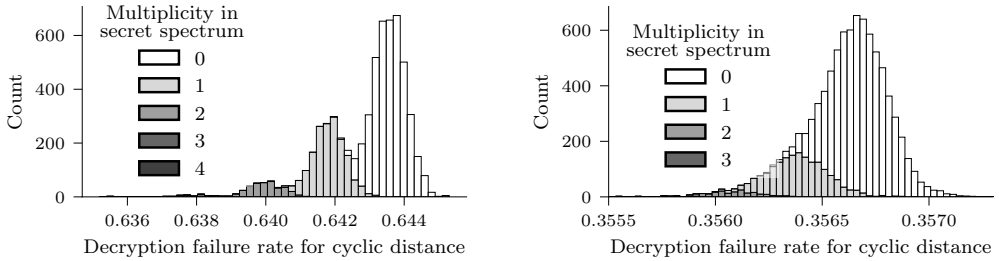


Figure 3. Effect of faults in BIKE Level 1 (Point B), and BIKE Level 5 (Point A).



(a) BIKE Level 1. **(b)** HQC Level 1.

Figure 4. Separation of distances in and out of the spectrum of (a) $\mathbf{h}_0$ (BIKE) after 3 million faulted decapsulations and (b) $\mathbf{y}$ (HQC) after 50 million faulted decapsulations.

that, if the Hamming weight of the secret syndrome is high enough, then the iteration will not have any effect, which may result in a decoding failure.

Figure 3 (left) shows the effect of this fault when applied to the second iteration of BIKE Level 1. We can see that a decryption failure or success faithfully characterizes the weight of the secret syndrome during the start of the BGF iteration when the fault is applied. We remark two important observations from our experiments. The first is that this fault location only gives meaningful information when applied to the second BGF iteration. Namely, when applied to the first iteration, we always get a decoding failure, and when applied after the second iteration, decoding always succeeded.

The second observation is that this fault location does not work for BIKE levels 3 and 5. This can be explained by the additional iterations required to support higher security levels, which makes it more robust. Nevertheless, we were able to find another skipping fault location that is applicable to BIKE levels 3 and 5: the subtraction in Lines 13 (C implementation) and 16 (Assembly) of Listing 4. The effect, shown in Figure 3, is noisier and less pronounced than the one from BIKE Level 1. However, decapsulation failures and successes are correlated with higher and smaller Hamming weights in such a way that our method is also applicable to the higher security levels.

Key recovery attack on BIKE Level 1 with BGF decoder. To validate our attack, we applied the key recovery procedure against BIKE Level 1. The spectrum recovery phase from decapsulation failure resulted in a sufficient separation between distances in and out of the spectrum after 3 million decapsulations, after which all 10 key recovery attacks were able to find the complete keys successfully (see Section 6 for the discussion on fault repeatability). The spectrum separation at 3 million decapsulation is shown in Figure 4a.

Key recovery attack on BIKE Level 1 with BIKE-Flip decoder. BIKE v5.2 recommends the BIKE-Flip decoder, which is not implemented in constant time (see Section 2.1).

```
1  void hqc_pke_encrypt(ciphertext_pke_t *c_pke, uint8_t *ek_pke, uint64_t *m, uint8_t *theta) {
2      ...
3      code_encode(c_pke->v, m);
4      vect_mul(tmp, r2, s);
5      vect_add(tmp, e, tmp, VEC_N_SIZE_64);
6      vect_truncate(tmp);
7      vect_add(c_pke->v, c_pke->v, tmp, VEC_N1N2_SIZE_64);
8      ...
9  }
10 void vect_add(uint64_t *o, const uint64_t *v1, const uint64_t *v2, uint32_t size) {
11   for (uint32_t i = 0; i < size; ++i)
12     o[i] = v1[i] ^ v2[i];
13 }
```

Listing 5. C implementation of the encryption procedure.

However, we simulate fault attacks on BIKE-Flip by assuming that some core functionalities are implemented similarly to BGF. First we noticed that BIKE-Flip is more robust to faulting the threshold computation because it runs for more iterations than BGF, resulting in no decapsulation failure. We then considered faults resulting in an early loop abort. While the attack was not successful using faults to reduce the number of BIKE-Flip iterations, we were able to attack BIKE-Flip by causing early aborts on the loop that performs the partial syndrome recomputation (Algorithm 1: Line 11). At 60% repeatability for loop abort faults (Section 6), key recovery needs 16.7 million decapsulations.

5.2 Fault attack on the vector addition in HQC reencryption

Our method makes the decryption a natural target for applying faults, as this is where the decoding is done. However, an interesting question that arises is whether it is possible to use our method to recover the key by faulting the re-encryption procedure. In this section, we show that this is indeed possible.

Fault location. Consider the encryption procedure from the point of view of an attacker who is crafting a ciphertext. Listing 5 shows the `hqc_pke_encrypt` procedure in HQC's reference implementation, which encrypts the message $\mathbf{m}$ and outputs the ciphertext $\mathbf{c} = (\mathbf{u}, \mathbf{v})$. Note that vector $\mathbf{v}$ is computed in Line 7, by calling function `vect_add`, which simply loops over all entries in the two vectors to add them together. Now suppose the attacker aborts the loop in Line 7 of `vect_add` in iteration $i^\star$. Then, the entries of the resulting faulted vector $\mathbf{v}'$ are defined as

$$\mathbf{v}'[i] = \begin{cases} \texttt{RMRS.Encode}(m)[i] + (\mathbf{s} \cdot \mathbf{r}_2)[i] + \mathbf{e}[i], & \text{if } 0 \le i \le i^\star, \\ \texttt{RMRS.Encode}(m)[i] & \text{if } i^\star < i < n. \end{cases}$$

Clearly the result $(\mathbf{u}, \mathbf{v}')$ would not be a valid ciphertext, and it would just cause a decapsulation failure due to the FO verification, as expected. However, if the attacker sends this ciphertext to the target and is able to use a fault injection during the reencryption to cause an early abort at iteration $i^\star$, then the target may accept the invalid ciphertext *even if the FO verification is protected*. This can be achieved with a single fault: after iteration $i^\star$, skip the conditional branch instruction that checks if the loop is over.

Applicability of our general attack method. Suppose the attacker sends the invalid ciphertext $\mathbf{c}' = (\mathbf{u}, \mathbf{v}')$ to the receiver. Then, the value computed by receiver during decryption is $\mathbf{v}' - \mathbf{y} \cdot \mathbf{u} = \texttt{RMRS.Encode}(\mathbf{m}) + \mathbf{e}_{\text{HQC}} + \mathbf{e}'$, where $\mathbf{e}'$ is the error associated with the early abort during the malicious encryption.

If the total error $\mathbf{e}_{\text{HQC}} + \mathbf{e}'$ has a sufficiently small Hamming weight, the receiver will still be able to recover $\mathbf{m}$ from the invalid ciphertext. But then the attacker who faults

the re-encryption will be able to learn if the decoding succeeded or not. Therefore, our general method should be applicable to this case. We just need to show that there exists a good value for $i^\star$ where the loop can be aborted to extract information from the faults.

Intuitively, $i^\star$ should stop the loop early enough (thus causing more errors) to allow little margin for the powerful Reed-Solomon decoder to always decode the message, but it cannot be too small otherwise decoding will always fail. In other words, a good $i^\star$ should make the decoding as sensible as possible to the original error $\mathbf{e}_{\text{HQC}}$, which is the one correlated with the secret key. While it is difficult to derive an optimal value analytically, we were able to experimentally find good values for $i^\star$ by iterating over its possible values, and selecting the one that yields a decapsulation failure probability closer to 50%.

Key recovery attack on HQC-128. For HQC-128, the optimal iteration to stop the loop was found to be $i^\star = 183$, where we observed that around 35% of the decapsulations failed. We ran key recovery attacks against 10 keys, and the spectrum recovery experiment was significantly noisier than what we observed for BIKE: for HQC-128 a spectrum separation necessary for key recovery was only observed at about 50 million challenges The separation is illustrated in Figure 4b. At that point, we were able to recover all 10 keys. Assuming 60% repeatability rate for loop abort fault (see Section 6), we would need about 83.3 million decapsulations to recover the key.

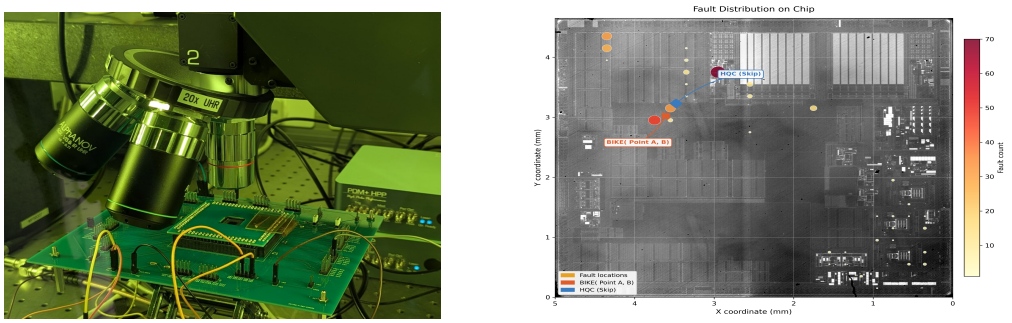
6 Experimental validation of fault models

Experimental Setup. We use Laser Fault Injection (LFI) as the primary attack vector in our experiments, as shown in Figure 5a. The LFI setup is based on a 980 nm diode laser capable of delivering up to 2 W of peak optical power, with a minimum spot size of 1.3 µm and configurable pulse width ranging from 1 nm and 4 µs. The device under test (DUT) is the STM32H753ZIT microcontroller, which integrates an ARM Cortex-M7 core (Armv7-M architecture). The setup further incorporates dedicated control software for precise trigger synchronization, an oscilloscope for validating pulse characteristics, and a relay-based automation mechanism for resetting the DUT between injections.

Threat Model. We assume that an attacker with direct physical access to the Device Under Test (DUT), capable of decapsulating the package without additional modifications, as is standard in laser fault injection scenarios. The attacker can monitor the timing characteristics of cryptographic operations to precisely trigger fault injection pulses. However, no capabilities are assumed for probing internal memory or register states.

Evaluation Results. Using the experiment setup, we performed the fault attacks described in Section 5, by attacking different target instructions for BIKE and HQC implementations, by carefully aligning the laser pulses. The STM32H753ZIT, as a 32-bit microcontroller widely used in embedded systems, provides a representative evaluation platform for fault injection attacks. The correctness of the induced fault was verified by comparing the input-output behavior between faulted and non-faulted executions. Table 4 summarizes the laser parameters and fault repeatability at various locations in BIKE and HQC.

Discussion. For attacking BIKE's BGF decoder, the fault attack configured with specific pulse width and pulse delay achieved 100% repeatability at point A and 99.6% repeatability at point B (Listing 4), with the fault-sensitive locations marked in red in Figure 5b. Therefore, the attack remains highly reliable, and successful key recovery as detailed in section 5.1 can be achieved with approximately 3 million fault injections in the worst case. In particular, the remaining 0.4% of missed faults are incorporated as noise, and



(a) (b)

Figure 5. The experimental setup (a) Laser Fault Injection (LFI) Setup and (b) Fault Sensitivity Scan on STM32H753ZIT.

Table 4. Practical Fault Validation on BIKE and HQC Implementations.

Implementation	Fault model	Pulse width	Pulse delay	Repeatability
BIKE	Point A (Listing 4)	35 ns	128 ns	100%
	Point B (Listing 4)	45 ns	705 ns	99.6%
HQC	Skip (Listing 5)	25 ns	259.3 µs	60%
	AND (Listing 3)	35 ns	128 ns	100%[1]

[1] Repeatability relies on dedicated trigger synchronization. Targeting multiple key bits may require per-bit synchronization and parameter tuning, reducing effective repeatability.

the key recovery procedure is robust enough to tolerate this noise, eliminating the need to explicitly filter out the missed faults.

In contrast, for the loop abort fault used against HQC (Listing 5) and the BIKE-Flip decoder, the observed repeatability is 60%. The remaining 40% of injections result in program crashes rather than incorrect outputs. Because crashes are easily detected, an attacker can simply repeat the fault injection process until a valid fault is achieved. While this makes the attack slightly less efficient than using an ideal fault oracle, the complexity only increases by a factor of $1/0.6 \approx 1.67$.

7 Mitigations and conclusions

Mitigations. One important and general attack surface unraveled by our work is the early loop abortion. This can be a major source of failures that introduce a controlled amount of error that perfectly suits our method. As such, it is critical that protected BIKE and HQC implementations perform loop integrity checks to ensure that all iterations were run. We recommend this to be done in all loops, which our preliminary tests indicate that would incur in a 3% overhead for decapsulation (considering HQC-128).

With respect to the BIKE decoder, we believe that simpler decoders like BIKE-Flip are beneficial. The regularity of BIKE-Flip, together with the greater number of iterations, makes it more appealing than BGF with respect to fault attacks: we could not perform attacks based on thresholds on it. For BGF, countermeasures based on threshold recomputation can become expensive (especially if the syndrome weight is recomputed). A more sensible countermeasure would be to enforce that the threshold lies within reasonable bounds. This is an inexpensive check, and should mitigate our fault attack that results in a threshold so high that no bit is flipped.

For HQC's polynomial multiplication, the best countermeasure seems to be the replica-

tion of the mask computation, followed by majority voting. This would not be as expensive as computing the mask on each internal loop, and should be effective against single-fault attacks. For HQC-128, we estimate an overhead lower than 0.1% for the decapsulation, when majority voting is implemented for the mask. We remark that shuffling countermeasures could be helpful if the shuffling is done with respect to the schoolbook multiplication blocks for Karatsuba. However, internal loop shuffling within a schoolbook multiplication may be ineffective: the attacker could still learn which 64-bit blocks of $\mathbf{y}$ contain a 1, which is critical information that can speed up key recovery using information-set decoding (ISD) algorithms with hints [HPR+22].

While these countermeasures should prevent the attacks presented in this paper, they may not be sufficient against more advanced attacks that inject more than one fault per decapsulation. Moreover, our general method can be used to find other exploitable fault locations that are not protected by the countermeasures above.

Limitations. The primary limitation of the proposed attacks is the relatively large number of successful fault injections required for key recovery. Reducing this requirement by identifying more effective fault locations or optimizing the fault injection strategy is an important direction for future work. Further, our attacks require precise synchronization with target instructions, imposing stringent timing and instrumentation requirements that, in our experiments, were only met in a laboratory setting. This requirement, although not uncommon for fault injection, can become more severe if the fault location must be adapted from one execution to another.

Conclusions. In this work, we demonstrated that fault injection attacks against BIKE and HQC are not limited to bypassing the Fujisaki-Okamoto verification. We not only showed specific points where faults can be exploited for key recovery, but we unraveled a novel and broader class of fault-sensitive locations. This class is characterized by faults that increase the Hamming weight of the errors to be corrected by the decoders, which can result in failures that correlate with the secret key. Our attacks achieve full key recovery under NIST Level 1 parameters, requiring 17392 decapsulation queries for HQC and approximately 3 million for BIKE. We validated all attacks via laser fault injection on an STM32 microcontroller running reference implementations.

While effective countermeasures can be designed for each attack presented in this work, our approach may expose additional fault targets that are harder to anticipate. Under the considered fault model, we did not observe ineffective faults [DEK+18], whether they can be exploited in code-based KEMs more broadly remains a general open question. Reducing the number of required faults by identifying more effective locations or optimizing the fault injection strategy is an important direction for future work, as is a deeper study of these attacks once constant-time BIKE-Flip implementations become available. On the analysis side, tightening the query count of the second HQC attack remains an open problem.

Acknowledgment

This work is supported in parts by National Research Foundation, Singapore, and Cyber Security Agency of Singapore under its National Cybersecurity Research & Development Programme (Development of Secured Components & Systems in Emerging Technologies through Hardware & Software Evaluation NRF-NCR25-DeSNTU-0001). Part of this work was funded by the following Brazilian research agencies: FAPESP (grant 2020/09850-0), CNPq (grant 307732/2023-1), and Coordenação de Aperfeiçoamento de Pessoal de Nível Superior – Brasil (CAPES) – (Finance Code 001). The authors used AI tools to refine language and improve writing quality; all technical content, analysis, and conclusions are entirely original.

References

[ABB+24] Nicolas Aragon, Paulo Barreto, Slim Bettaieb, Loïc Bidoux, Olivier Blazy, Jean-Christophe Deneuville, Phillipe Gaborit, Santosh Ghosh, Shay Gueron, Tim Güneysu, Carlos Aguillar-Melchor, Rafael Misoczki, Edoardo Persichetti, Jan Richter-Brockmann, Nicolas Sendrier, Jean-Pierre Tillich, Valentin Vasseur, and Gilles Zémor. BIKE: Bit Flipping Key Encapsulation - Round 4 submission, October 2024. Available online at https://bikesuite.org/files/v5.2/BIKE_Spec.2024.10.10.1.pdf, last accessed on 03.30.2026. Cited on pages 1, 4, and 5.

[ABC+22] Martin R. Albrecht, Daniel J. Bernstein, Tung Chou, Carlos Cid, Jan Gilcher, Tanja Lange, Varum Maram, Ingo von Maurich, Rafael Misoczki, Ruben Niederhagen, Kenneth G. Patterson, Edoardo Persichetti, Christiane Peters, Peter Schwabe, Nicolas Sendrier, Jakub Szefer, Cen Jung Tjhai, Martin Tomlinson, and Wen Wang. Classic McEliece: conservative code-based cryptography, 2022. Available online at: classic.mceliece.org/nist/mceliece-submission-20221023.pdf, last accessed on 03.30.2026. Cited on page 1.

[ABD+21] Roberto Avanzi, Joppe W. Bos, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, John Schanck, Peter Schwabe, Gregor Seiler, and Damien Stehlé. CRYSTALS-Kyber: Algorithm specifications and supporting documentation (version 3.02), 2021. https://pq-crystals.org/kyber/data/kyber-specification-round3-20210804.pdf. Cited on page 1.

[CCD+21] Pierre-Louis Cayrel, Brice Colombier, Vlad-Florin Drăgoi, Alexandre Menu, and Lilian Bossuet. Message-recovery laser fault injection attack on the Classic McEliece cryptosystem. In Anne Canteaut and François-Xavier Standaert, editors, *Advances in Cryptology – EUROCRYPT 2021*, volume 12697 of *Lecture Notes in Computer Science*, pages 438–467, Cham, 2021. Springer International Publishing. https://doi.org/10.1007/978-3-030-77886-6_15. Cited on page 2.

[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, Aug. 2018. https://doi.org/10.13154/tches.v2018.i3.547-572. Cited on page 21.

[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, Aug. 2022. https://doi.org/10.46586/tches.v2022.i4.637-660. Cited on page 2.

[DGK] Nir Drucker, Shay Gueron, and Dusan Kostic. Additional implementation of BIKE (Bit Flipping Key Encapsulation). https://github.com/awslabs/bike-kem (accessed 2026-10-13). Cited on page 5.

[DGK20] Nir Drucker, Shay Gueron, and Dusan Kostic. QC-MDPC decoders with several shades of gray. In *Post-Quantum Cryptography – PQCrypto 2020*, volume 12100 of *Lecture Notes in Computer Science*, pages 35–50, Cham, 2020. Springer International Publishing. https://doi.org/10.1007/978-3-030-44223-1_3. Cited on page 5.

[FHS+17] Tomáš Fabšič, Viliam Hromada, Paul Stankovski, Pavol Zajac, Qian Guo, and Thomas Johansson. A reaction attack on the QC-LDPC McEliece cryptosystem. In Tanja Lange and Tsuyoshi Takagi, editors, *Post-Quantum Cryptography*, pages 51–68, Cham, 2017. Springer International Publishing. https://doi.org/10.1007/978-3-319-59879-6_4. Cited on page 2.

[FO99] Eiichiro Fujisaki and Tatsuaki Okamoto. Secure integration of asymmetric and symmetric encryption schemes. In *Advances in Cryptology – CRYPTO 1999*, volume 1666 of *Lecture Notes in Computer Science*, pages 537–554. Springer, 1999. https://doi.org/10.1007/3-540-48405-1_34. Cited on page 4.

[FO13] Eiichiro Fujisaki and Tatsuaki Okamoto. Secure integration of asymmetric and symmetric encryption schemes. *Journal of Cryptology*, 26(1):80–101, Jan 2013. https://doi.org/10.1007/s00145-011-9114-1. Cited on page 1.

[Gal62] Robert Gallager. Low-density parity-check codes. *IRE Transactions on Information Theory*, 8(1):21–28, 1962. https://doi.org/10.1109/TIT.1962.1057683. Cited on page 6.

[GAMA+25] Phillipe Gaborit, Carlos Aguillar-Melchor, Nicolas Aragon, Loïc Bidoux, Olivier Blazy, Jean-Christophe Deneuville, Edoardo Persichetti, Gilles Zémor, Jurjen Bos, Aranaud Dion, Jérôme Lacan, Jean-Marc Robert, Pascal Véron, Paulo Barreto, Santosh Ghosh, Shay Gueron, Tim Güneysu, Rafael Misoczki, Jan Richter-Brockmann, Nicolas Sendrier, Jean-Pierre Tillich, and Valentin Vasseur. Hamming Quasi-Cyclic (HQC) Specification, 2025. Available online at https://pqc-hqc.org/doc/hqc_specifications_2025_08_22.pdf. Cited on pages 1, 6, and 7.

[GJ20] Qian Guo and Thomas Johansson. A new decryption failure attack against HQC. In Shiho Moriai and Huaxiong Wang, editors, *Advances in Cryptology – ASIACRYPT 2020*, pages 353–382, Cham, 2020. Springer International Publishing. https://doi.org/10.1007/978-3-030-64837-4_12. Cited on page 2.

[GJS16] Qian Guo, Thomas Johansson, and Paul Stankovski. A key recovery attack on MDPC with CCA security using decoding errors. In Jung Hee Cheon and Tsuyoshi Takagi, editors, *Advances in Cryptology – ASIACRYPT 2016*, volume 10031 of *Lecture Notes in Computer Science*, pages 789–815, Berlin, Heidelberg, 2016. Springer Berlin Heidelberg. https://doi.org/10.1007/978-3-662-53887-6_29. Cited on pages 3, 13, and 14.

[HGS99] Chris Hall, Ian Goldberg, and Bruce Schneier. Reaction attacks against several public-key cryptosystems. In Vijay Varadharajan and Yi Mu, editors, *Information and Communication Security – ICICS 1999*, volume 1726 of *Lecture Notes in Computer Science*, pages 2–12, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg. https://doi.org/10.1007/978-3-540-47942-0_2. Cited on page 2.

[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, Cham, 2017. Springer International Publishing. https://doi.org/10.1007/978-3-319-70500-2_12. Cited on pages 1 and 4.

[HMS+23] Julius Hermelink, Erik Mårtensson, Simona Samardjiska, Peter Pessl, and Gabi Dreo Rodosek. Belief propagation meets lattice reduction: Security estimates for error-tolerant key recovery from decryption errors. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2023(4):287–317, 2023. https://doi.org/10.46586/tches.v2023.i4.287-317. Cited on page 2.

[HPP21] Julius Hermelink, Peter Pessl, and Thomas Pöppelmann. Fault-Enabled Chosen-Ciphertext Attacks on Kyber. In *Progress in Cryptology – INDOCRYPT 2021*, volume 13143 of *Lecture Notes in Computer Science*, pages 311–334. Springer, 2021. https://doi.org/10.1007/978-3-030-92518-5_15. Cited on page 2.

[HPR+22] Anna-Lena Horlemann, Sven Puchinger, Julian Renner, Thomas Schamberger, and Antonia Wachter-Zeh. Information-set decoding with hints. In Antonia Wachter-Zeh, Hannes Bartz, and Gianluigi Liva, editors, *Code-Based Cryptography*, pages 60–83, Cham, 2022. Springer International Publishing. https://doi.org/10.1007/978-3-030-98365-9_4. Cited on page 21.

[KPR+] Matthias J. Kannwischer, Richard Petri, Joost Rijneveld, Peter Schwabe, and Ko Stoffelen. PQM4: Post-quantum crypto library for the ARM Cortex-M4. https://github.com/mupq/pqm4. Cited on page 3.

[Nat24] National Institute of Standards and Technology. Module-Lattice-Based Key-Encapsulation Mechanism Standard. Technical Report Federal Information Processing Standards Publications (FIPS) 203, U.S. Department of Commerce, Washington, D.C., 2024. https://doi.org/10.6028/nist.fips.203. Cited on page 1.

[PGD+23] Sabine Pircher, Johannes Geier, Julian Danner, Daniel Mueller-Gritschneder, and Antonia Wachter-Zeh. Key-recovery fault injection attack on the Classic McEliece KEM. In Jean-Christophe Deneuville, editor, *Code-Based Cryptography – CBCrypto 2022*, volume 13839 of *Lecture Notes in Computer Science*, pages 37–61, Cham, 2023. Springer Nature Switzerland. https://doi.org/10.1007/978-3-031-29689-5_3. Cited on page 2.

[PP21] Peter Pessl and Lukáš Prokop. Fault attacks on CCA-secure lattice kems. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2021(2):37–60, 2021. https://doi.org/10.46586/tches.v2021.i2.37-60. Cited on page 2.

[PT18] Thales B. Paiva and Routo Terada. Improving the efficiency of a reaction attack on the QC-MDPC McEliece. *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, E101.A(10):1676–1686, 2018. https://doi.org/10.1587/transfun.E101.A.1676. Cited on page 2.

[PT20] Thales B. Paiva and Routo Terada. A timing attack on the HQC encryption scheme. In Kenneth G. Paterson and Douglas Stebila, editors, *Selected Areas in Cryptography – SAC 2019*, volume 11959 of *Lecture Notes in Computer Science*, pages 551–573, Cham, 2020. Springer International Publishing. https://doi.org/10.1007/978-3-030-38471-5_22. Cited on pages 3, 13, and 14.

[PT22] Thales B. Paiva and Routo Terada. Faster constant-time decoder for MDPC codes and applications to BIKE KEM. *IACR Transactions on Cryptographic*

Hardware and Embedded Systems, 2022(4):110–134, Aug. 2022. https://doi.org/10.46586/tches.v2022.i4.110-134. Cited on page 16.

[RCDB24] 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*, 23(2):35:1–35:54, 2024. https://doi.org/10.1145/3603170. Cited on page 2.

[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 *Constructive Side-Channel Analysis and Secure Design – COSADE 2019*, volume 11421 of *Lecture Notes in Computer Science*, pages 232–250. Springer, 2019. https://doi.org/10.1007/978-3-030-16350-1_13. Cited on page 2.

[SM17] Douglas Stebila and Michele Mosca. Post-quantum key exchange for the internet and the Open Quantum Safe Project. In Roberto Avanzi and Howard Heys, editors, *Selected Areas in Cryptography – SAC 2016*, volume 10532 of *Lecture Notes in Computer Science*, pages 14–37, Cham, 2017. Springer International Publishing. https://doi.org/10.1007/978-3-319-69453-5_2. Cited on page 3.

[SSPB19] Simona Samardjiska, Paolo Santini, Edoardo Persichetti, and Gustavo Banegas. A reaction attack against cryptosystems based on LRPC codes. In Peter Schwabe and Nicolas Thériault, editors, *Progress in Cryptology – LATINCRYPT 2019*, volume 11774 of *Lecture Notes in Computer Science*, pages 197–216, Cham, 2019. Springer International Publishing. https://doi.org/10.1007/978-3-030-30530-7_10. Cited on page 2.

[SSV26] Magali Salom, Nicolas Sendrier, and Valentin Vasseur. Sparse vector reconstruction from distance spectrum using soft information. In Magali Bardet and Ruben Niederhagen, editors, *Post-Quantum Cryptography – PQCrypto 2026*, volume 16491 of *Lecture Notes in Computer Science*, pages 99–129, Cham, 2026. Springer Nature Switzerland. https://doi.org/10.1007/978-3-032-22695-2_4. Cited on page 14.

[VRBW25] Hariprasad Kelassery Valsaraj, Prasanna Ravi, Shivam Bhasin, and Hongjun Wu. When randomness isn't random: Practical fault attack on post-quantum lattice standards. Cryptology ePrint Archive, Paper 2025/2009, 2025. https://eprint.iacr.org/2025/2009. Cited on page 2.

[XIU+21] Keita Xagawa, Akira Ito, Rei Ueno, Junko Takahashi, and Naofumi Homma. Fault-injection attacks against NIST's post-quantum cryptography round 3 KEM candidates. In *Advances in Cryptology – ASIACRYPT 2021*, volume 13091 of *Lecture Notes in Computer Science*, pages 33–61. Springer, 2021. https://doi.org/10.1007/978-3-030-92075-3_2. Cited on pages 2, 3, and 12.