Fault Attacks on CCA-secure Lattice KEMs

Peter Pessl[1†] and Lukas Prokop[2]

[1] Infineon Technologies, Germany
peter@pessl.cc
[2] Graz University of Technology, Austria
lukas.prokop@iaik.tugraz.at

Abstract. NIST's post-quantum standardization effort very recently entered its final round. This makes studying the implementation-security aspect of the remaining candidates an increasingly important task, as such analyses can aid in the final selection process and enable appropriately secure wider deployment after standardization. However, lattice-based key-encapsulation mechanisms (KEMs), which are prominently represented among the finalists, have thus far received little attention when it comes to fault attacks.

Interestingly, many of these KEMs exhibit structural similarities. They can be seen as variants of the encryption scheme of Lyubashevsky, Peikert, and Rosen, and employ the Fujisaki-Okamoto transform (FO) to achieve CCA2 security. The latter involves re-encrypting a decrypted plaintext and testing the ciphertexts for equivalence. This corresponds to the classic countermeasure of computing the inverse operation and hence prevents many fault attacks.

In this work, we show that despite this inherent protection, practical fault attacks are still possible. We present an attack that requires a single instruction-skipping fault in the decoding process, which is run as part of the decapsulation. After observing if this fault actually changed the outcome (effective fault) or if the correct result is still returned (ineffective fault), we can set up a linear inequality involving the key coefficients. After gathering enough of these inequalities by faulting many decapsulations, we can solve for the key using a bespoke statistical solving approach. As our attack only requires distinguishing effective from ineffective faults, various detection-based countermeasures, including many forms of double execution, can be bypassed.

We apply this attack to Kyber and NewHope, both of which belong to the aforementioned class of schemes. Using fault simulations, we show that, e.g., 6,500 faulty decapsulations are required for full key recovery on Kyber512. To demonstrate practicality, we use clock glitches to attack Kyber running on a Cortex M4. As we argue that other schemes of this class, such as Saber, might also be susceptible, the presented attack clearly shows that one cannot rely on the FO transform's fault deterrence and that proper countermeasures are still needed.

Keywords: lattice-based cryptography, key encapsulation, fault attack

1 Introduction

The search for quantum secure replacements of RSA and DLP-based cryptosystems is in full swing. This is demonstrated by NIST's ongoing Post-Quantum Cryptography Standardization process [NIS16], which very recently entered its third and final round. In this selection effort, a large number of submitted proposals base their security arguments on lattice problems. Such lattice-based key-encapsulation mechanisms (KEMs) and digital

[†]Part of this work was done while Peter Pessl worked at Graz University of Technology.

signature schemes offer both comparatively small public key sizes and highly competitive runtimes. This makes them prime candidates for applications in embedded systems and constrained devices.

Exactly these devices are highly susceptible to implementation attacks, such as power analysis and fault injections. Thus, analyzing potential side-channel vulnerabilities and respective countermeasures is an important task, as also stated by NIST. In this context, it is interesting that many lattice-based KEMs have lots of similarities, at least on higher abstraction levels. For this reason, attacks exploiting such high-level properties are potentially applicable to a broad set of schemes.

More concretely, many lattice-based KEMs, e.g., Kyber [ABD+19], NewHope [PAA+19], and Saber [DKRV19], can be seen as variants of the public-key encryption scheme presented by Lyubashevsky, Peikert, and Regev (LPR) [LPR10]. As plain LPR only provides security against chosen-plaintext attacks (CPA), all of the mentioned schemes employ the Fujisaki-Okamoto (FO) transform [FO99], which allows building a CCA-secure KEM using a CPA-secure public-key encryption scheme. Simply speaking, the FO transform re-encrypts each plaintext after decryption and then tests for equality of the received ciphertext and the outcome of the re-encryption. If this test fails, then the plaintext (or shared key) is discarded.

This re-encryption after decryption corresponds to computing an inverse operation and thus can also be seen as an instance of a classic countermeasure against fault attacks. When injecting a fault into the decryption and thereby changing the computed plaintext, the equality test after re-encryption will fail. The faulty plaintext is not released, thereby thwarting many fault attacks. This raises the question if such CCA-secure lattice-KEMs can still be attacked using relatively simple faults.

Our Contribution. In this work, we answer this question positively. That is, we present a key-recovery fault attack requiring many faulted decapsulations, but only injecting a single instruction skip per decapsulation call. Crucially, we do not attack the equality test, which is an obvious faulting target and thus clearly requires protection.

Instead, we target the decoding process, which is required for removing noise from the decrypted plaintext. We skip a single instruction in the decoder and then observe if this fault leads to a decoding error and is thus detected by the re-encryption process (effective fault), or if the correct plaintext is still computed despite the injected skip (ineffective fault). This single bit of information can be used to construct a linear inequality over the key. We present an algorithm that can solve for the private key, given enough such inequalities. As we only need to discern effective from ineffective faults, our attack is not impeded by some basic fault countermeasures, such as (course-grained) double execution.

We apply our attack to Kyber as well as NewHope and show that masked implementations can also still be vulnerable. When attacking the smallest Kyber parameter set, for instance, our attack requires faulting 6,500 decryptions. We also perform the attack on Kyber running on an ARM-based microcontroller, thereby demonstrating practicality.

Outline. In Section 2, we recall the LPR cryptosystem and the Fujisaki-Okamoto transform. This is followed by a generic description of Kyber and NewHope. In Section 3, we describe our attack on the example of a generic LPR-based KEM. Section 4 then applies the attack on Kyber, NewHope, and a masked decoder implementation. In Section 5, we evaluate the attack performance using simulations, i.e., we give a success rate and analyze how many faults are required for full key recovery. In Section 6, we then back up our claims by attacking Kyber using clock glitches. Finally, we discuss possible future work and conclude in Section 7.

Algorithm 1 LPR Key Generation

Output: Keypair (pk, sk)
1: $s, e \in \mathcal{R}_q \leftarrow \chi^n$
2: $a \leftarrow \mathcal{U}_q^n$
3: $b = as + e$
4: **return** $(pk = (a, b), sk = (a, s))$

Algorithm 2 LPR Encryption

Input: Public key $pk = (a, b)$, n-bit message m
Output: Ciphertext (u, v)
1: $r, e_1, e_2 \in \mathcal{R}_q \leftarrow \chi^n$
2: $u = ar + e_1$
3: $v = br + e_2 + m \cdot \lfloor q/2 \rfloor$
4: **return** (u, v)

2 Background

In the following, $[a, b)$ represents an open-closed interval, whereas $[a, b]$ includes both boundary values. We denote vectors and polynomials using lower-case characters, like s, individual coefficients of s are written as $s[i]$. Upper-case symbols like A denote matrices. Multiplication of polynomials a, b in some ring $\mathcal{R}$ is denoted as $a \cdot b$ or simply ab. (Coefficient-wise) sampling from a probability distribution is represented with $\leftarrow$, deterministic assignments with $=$, and equality tests with $==$. This corresponds to the C programming language, which is used in all following source code excerpts.

2.1 LPR Public-Key Encryption

In 2010, Lyubashevsky, Peikert, and Regev [LPR10] extended Regev's Learning With Errors (LWE) problem [Reg05] to an algebraic variant called Ring-LWE (RLWE). They also defined a public-key encryption scheme, frequently dubbed the LPR scheme, which offers a reduction to RLWE.

The LPR cryptosystem uses polynomials in the ring $\mathcal{R}_q = \mathbb{Z}_q[x]/(x^n + 1)$, with a prime q and n a power of two. The parameters q and n are typically chosen such that polynomial multiplication can be performed in $\mathcal{O}(n \log n)$ using the Number Theoretic Transform (NTT). LPR requires sampling from a (narrow) error distribution χ. Early proposals, such as the original LPR scheme, used a discrete Gaussian distribution for χ. However, Gaussian samplers are hard to implement securely. For this reason, many newer proposals, including Kyber and NewHope, use a centered binomial distribution over the range $[-\eta, \eta]$ for some small integer η.

The encryption scheme is comprised of three procedures. Algorithm 1 (Key Generation) generates a public/secret key pair. The coefficients of the secret key s and the noise polynomial e are drawn from χ, the coefficients of a are sampled from $\mathcal{U}_q$, i.e., from the uniform distribution over $\mathbb{Z}_q$. The public key b is computed as $b = as + e$. While only s is stored as part of the secret key, we note that e must also be considered a secret, as knowing its value would allow trivial recovery of s. Encryption is shown in Algorithm 2. In its first step, three polynomials r, e_1, e_2 are sampled from the error distribution. Then, the two ciphertext components u, v are computed. For computing the second component v, the n-bit message m needs to be mapped to a polynomial in $\mathcal{R}_q$. The simplest way of doing that is by multiplying each bit of m with $\lfloor q/2 \rfloor$. Finally, Algorithm 3 specifies the decryption step. Its correctness can be verified by back-substitution:

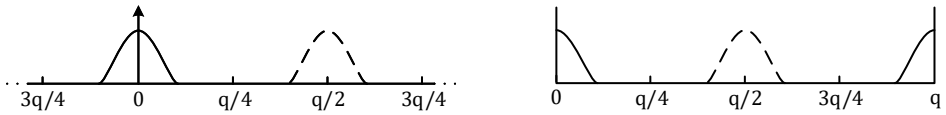
Algorithm 3 LPR Decryption

Input: Secret key $sk = (a, s)$, ciphertext (u, v)
Output: Message m
1: $m' = v - us$
2: **return** DECODE(m')

$$\begin{aligned} v - us &= br + e_2 + m \cdot \left\lfloor \frac{q}{2} \right\rfloor - (ar + e_1)s \\ &= asr + er + e_2 + m \cdot \left\lfloor \frac{q}{2} \right\rfloor - ars - e_1 s \\ &= m \cdot \left\lfloor \frac{q}{2} \right\rfloor + er + e_2 - e_1 s \end{aligned} \tag{1}$$

As seen above, computing $m' = v - us$ yields a *noisy* version of the plaintext. We call the added terms, i.e., $er + e_2 - e_1 s$, the *encryption noise* d. As all terms involved in this encryption noise are sampled from the narrow error distribution χ, the original message bits can still be recovered. This can be seen in Figure 1, which shows an exemplary probability distribution for the coefficients of the noisy plaintext in two different ranges. The probability distribution of a 0 bit is shown with a solid line; the dashed line shows the distribution for a 1 bit. We use this mapping throughout the rest of this paper.

A dedicated decoding routine DECODE is needed to recover the original message m from m'. We defer the description of efficient constant-time decoder implementations to later sections.



(a) Range: $[-q/4, 3q/4)$ (b) Range: $[0, q)$

Figure 1: Typical probability distribution of the coefficients of the noise plaintext m'. The solid line marks the distribution for a 0 bit, the dashed line for a 1 bit.

2.2 Fujisaki-Okamoto Transform

The LPR public-key encryption scheme only offers security against chosen-plaintext attacks (CPA). Thus, LPR key pairs must be ephemeral, as the plain scheme can be trivially broken in the chosen-ciphertext setting (CCA) if keys are reused (see, e.g., [Flu16]). For this reason, many LPR-based schemes make use of the Fujisaki-Okamoto (FO) transform [FO99] or one of its more recent versions [TU16, HHK17]. This transform allows constructing a CCA-secure key-exchange mechanism (KEM) using a CPA-secure public-key encryption scheme.

The employed FO variants are, at least from a high-level perspective, very similar. To illustrate the basic principles of the transform, we now give more details on the FO variant used by NewHope [PAA+19]. NewHope refers to its key-encapsulation transformation as $\text{QFO}_m^{\not\perp\prime}$. It embeds a public-key encryption scheme PKE featuring key generation, encryption, and decryption. Crucially, the randomness required for encryption is made explicit via a seed parameter *coin*. Furthermore, let F and G be two hash functions. Figure 2 shows the full $\text{QFO}_m^{\not\perp\prime}$ process. In ENCAPS, a randomly chosen message μ is encrypted using a seed derived via G. The shared secret ss is computed by hashing

KEYGEN()	ENCAPS(pk)	DECAPS($(c,d),(sk,pk,s)$)

$(pk, sk) = \text{PKE.KeyGen}()$
$s \leftarrow \{0,\ldots,255\}^{\text{len}_s}$
$\overline{sk} = (sk, pk, s)$
return $(pk, \overline{sk})$
	$\mu \leftarrow \mathcal{M}$
	$(K, \text{coin}', d) = G(pk \Vert \mu)$
	$c = \text{PKE.Encrypt}(pk, \mu; \text{coin}')$
	$\overline{c} = (c, d)$
	$ss = F(K \Vert c \Vert d)$
	return $(\overline{c}, ss)$
		$\mu' = \text{PKE.Decrypt}(c, sk)$
		$(K', \text{coin}'', d') = G(pk \Vert \mu')$
		$c' = \text{PKE.Encrypt}(pk, \mu'; \text{coin}'')$
		if $c == c'$ **and** $d == d'$ **then**
		  return $ss' = F(K' \Vert c \Vert d)$
		else
		  return $ss' = F(s \Vert c \Vert d)$

Figure 2: IND-CCA transform $\text{QFO}_m^{\not\perp\prime}$ built using public key encryption scheme PKE and hash functions F and G [PAA+19].

the PKE-ciphertext c together with the confirmation value d. Decapsulation recovers the message μ and then re-encrypts it using the recovered seed *coin*. Only if both the PKE-ciphertexts as well as the confirmation values d match, the correct shared secret is returned. Upon failure, a pseudorandom shared secret is computed.

While the intended purpose of the FO is to establish CCA security, it also averts many fault attacks by design. For instance, when targeting PKE.DECRYPT in DECAPS, i.e., the only part involving the private key sk, all faults leading to a corrupted μ' are detected, as re-encryption will lead to a different c'. This rules out differential fault attacks exploiting resulting differences in μ'.

2.3 Kyber

Kyber [ABD+19] is one of four finalists in the KEM category of the NIST PQC standardization process. It is an example of a scheme combining an LPR-like public-key encryption scheme with the FO to construct a CCA-secure KEM. The Kyber second-round specification includes three parameter sets (Kyber512, Kyber768, and Kyber1024), which are listed in Table 2 of Appendix A.

Structure and differences. Unlike LPR, Kyber bases its security on the Learning with Errors problem in module lattices (Module-LWE). This means that, e.g., a is not a polynomial, but instead a quadratic matrix of polynomials $A \in \mathcal{R}_q^{k \times k}$, where k is the module rank. Other elements, such as the secret key s, the error e, as well as r and e_1, are now (column) vectors containing polynomials, i.e., $\mathcal{R}_q^k$. Kyber uses the same base ring $\mathcal{R}_q$ for all its parameter sets, only the module rank k varies. All parameters are chosen such that polynomial multiplication can be efficiently implemented using a slightly modified NTT.

Another difference to plain LPR is the use of (lossy) ciphertext compression. In compression, a value x is divided by $q/2^d$, with d being the compression parameter, and then rounded to the nearest integer. Decompression reverses this process, but can of course only recover an approximation to x. In simplified terms, only the d most significant bits of x are kept, all lower bits are discarded. By doing that, the size of the ciphertext can be significantly decreased. Kyber compresses both ciphertext components (u, v), but to a different degree. The two compression parameters (d_u, d_v) are specified in the parameter set.

Message encoding and decoding. For message encoding, i.e., mapping a message to an element in $\mathcal{R}_q$, Kyber follows the path of LPR and multiplies each message bit with $\lfloor q/2 \rfloor$. Recall that for decryption, a decoder is needed to recover the original message m from its noisy version m'. This decoder is applied to each coefficient of m' and returns 1 if the input value is in $[q/4, 3q/4)$, and 0 otherwise. This must be done in a secure (read:

constant time) manner, otherwise side-channel attackers might be able to infer information on m or s.

The decoding routine of Kyber's reference implementation[1] is sketched in Figure 3. First, it assures that all coefficients of the input polynomial a are in $[0, q)$. Then, it loops over all coefficients (byte index i, bit index j). Decoding as such is then done in a single line of code, where all values are interpreted as unsigned integers. After multiplying the coefficient of the input a by 2 and adding $\lfloor q/2 \rfloor$, an integer division by q is performed. Note that compilers turn such divisions with constants into multiplications[2], thereby avoiding typically non-constant-time division operations. Finally, the least-significant bit of the outcome gives the decoded message bit.

```
1  uint16_t t = (((a->coeffs[8*i+j] << 1) + KYBER_Q/2) / KYBER_Q) & 1;
2  msg[i] |= t << j;
```

Figure 3: Kyber's `poly_tomsg` function converts a polynomial a to a 32-byte message `msg`. After ensuring each coefficient of a lies within $[0, q)$, the first line maps each coefficient to 0 or 1. Here, $0 \leq i \leq 31$ and $0 \leq j \leq 7$ yield all coefficient indices.

To further illustrate this process, in Figure 4, we show how each of the above operations affects the probability distribution of the intermediate value (also cf. Figure 1). The multiplication by 2 scales the x axis, the following addition shifts the distributions to the right. The integer division by q leads to values between 0 and 2, picking the LSB then gives the correct decoded bit.

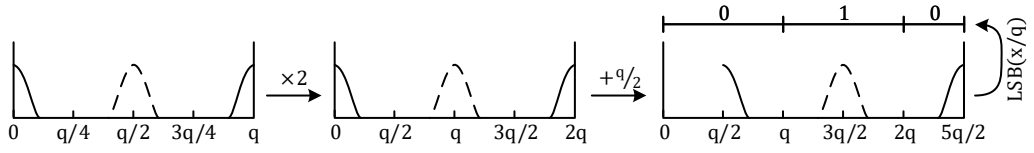


Figure 4: Visualization of the decoding routine used in Kyber's reference implementation.

2.4 NewHope

NewHope [PAA+19] is a key-encapsulation mechanism which bases its security on RLWE. While it did not advance to the final round of the PQC process, it is still of significance, as it has already undergone some first real-world evaluations [Lan16, Inf17]. Just like Kyber, it uses an LPR-like public-key encryption scheme in combination with the FO described in Section 2.2 to construct a CCA-secure KEM. The NewHope NIST submission includes two parameter sets, with $n = 512$ and $n = 1024$, respectively. We give all other specified values in Table 3 of Appendix A.

Structure and differences. NewHope is, compared to Kyber, arguably a more direct successor to LPR due to it also being based on RLWE. Thus, it does not operate with matrices and vectors of polynomials. NewHope also employs ciphertext compression, but unlike Kyber, it only compresses v.

[1] https://github.com/pq-crystals/kyber

[2] In our experiments, gcc always represented divisions by a constant as multiplication whereas LLVM used this substitution only in optimized builds. This was tested with various gcc (8.2–10.2) and clang (10–11) versions on several platforms using godbolt.org

Message encoding and decoding routine. A distinguishing feature of NewHope is its message encoding routine. As encrypted messages are 256 bits long, whereas the used ring dimension n is much larger, it encodes each bit onto multiple coefficients and thereby increases resilience against decryption errors. Concretely, it repeats the message twice (for $n = 512$) or four times (for $n = 1024$) and then multiplies the coefficients of this extended message with $\lfloor q/2 \rfloor$.

The decoding routine first applies the function $\text{FLIPABS}(x) = |(x \bmod q) - q/2|$ to all coefficients. We show in Figure 5 how FLIPABS affects the coefficient-wise probability distribution of $v - us$; we also give its C code as included in NewHope's reference implementation[3] in Figure 6.

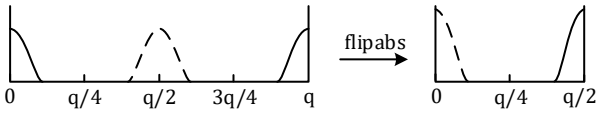


Figure 5: NewHope FLIPABS routine

The function `coeff_freeze` reduces the input x into the base interval $[0, q)$. The subsequent subtraction sets r's sign bit if $x < q/2$. Recognize that r is declared as a signed integer. Thus, the following line performs an *arithmetic* shift by 15 positions. If r is positive (sign bit is 0), this shift flushes out all set bits and yields 0. Shifting a negative r (sign bit is 1) leads to all bits being set to 1, i.e., `0xffff`, which equals -1 for an `int16_t`.

This property can be used in line 8. If m is zero, then both the addition and the XOR have no effect. If m is `0xffff`$= -1$, all bits are flipped after subtracting 1. This is exactly the process of computing the two's complement. Thus, lines 7 and 8 perform a conditional negation and thereby compute the absolute value of r.

```
1  static uint16_t flipabs(uint16_t x)
2  {
3    int16_t r,m;
4    r = coeff_freeze(x);
5
6    r = r - NEWHOPE_Q/2;
7    m = r >> 15;
8    return (r + m) ^ m;
9  }
```

Figure 6: C source for the FLIPABS function of NewHope's reference implementation. This function computes $|(x \bmod q) - \frac{q}{2}|$.

After computing FLIPABS, the decoding routine sums up the 2 (NewHope512) or 4 (NewHope1024) coefficients encoding the same message bit. If this sum is larger than either $q/2$ (for $n = 512$ and two coefficients per message bits) or q (for $n = 1024$), then the decoder returns 1, and 0 otherwise. This comparison can be performed in constant time by subtracting $q/2$ (or q, respectively) and then returning the MSB.

2.5 Masked Decoder

The previously mentioned applicability of lattice-based schemes to embedded systems raises the need for adequate protection against side-channel attacks. Due to the linearity of many involved operations, e.g., polynomial addition and multiplication in $\mathcal{R}_q$, the masking

[3] https://github.com/newhopecrypto/newhope

countermeasure appears to be a perfect fit. Extending masking to the nonlinear decoder, however, is a much more involved task.

One possible way of building such a masked decoder was presented by Oder et al. [OSPG18], who describe a fully masked implementation of an LPR-like lattice KEM. Similar to NewHope, their custom KEM encodes each message bit onto multiple polynomial coefficients. The accompanying masked decoder is designed to handle this fact. As our first target is Kyber, we now describe the simpler version geared towards the single-coefficient-per-bit encoding scheme. Also, we use the Kyber parameter set, with $q = 3329$ and $\lceil \log_2 q \rceil = 12$, for all further illustrations.

The decoding routine is illustrated in Figure 7. This figure again shows how each operation performed by the decoding algorithm affects the intermediate value's probability distribution. As all operations are performed in a modular domain (either modulo q or modulo a power of two), we adapt the circular representation of Oder et al. Note that the figure shows the distribution of the unmasked value, but all operations are performed in the masked domain. That is, the decoder is fed each coefficient m' in masked form, i.e., the two shares m'_1 and m'_2 satisfy $m' = m'_1 + m'_2$ in $\mathbb{Z}_q$. All subsequent operations are also done in a masked manner.

As a first step, the decoding routine subtracts $q/4$ from the input coefficient. Since this is a linear operation in $\mathbb{Z}_q$, it suffices to perform this subtraction on just one of the two shares. Then, the shares are transformed, from arithmetically mod q to arithmetically mod 2^{bits}, where $\text{bits} = \lceil \log_2 q \rceil + 1$. For the details of this transform, we refer to the original paper [OSPG18]. After subtracting $q/2 \pmod{2^{\text{bits}}}$ from one of the shares and performing a final arithmetic-to-boolean mask conversion, a masked representation of the decoded message bit can be retrieved by selecting the MSBs of the shares.

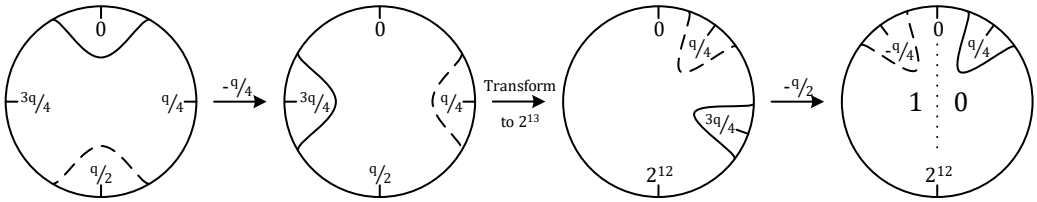


Figure 7: Visualization of the masked decoder of Oder et al. [OSPG18]

2.6 Previous Fault Attacks on Lattice-based KEMs

Thus far, the susceptibility of lattice-based KEMs against fault attacks is somewhat unexplored. Still, there exists some prior work.

Ravi et al. [RRB+19] propose a fault attack that can target key generation and encapsulation, but not decapsulation. They exploit the fact that long secrets are generated by expanding a short seed, which is additionally used multiple times but with a different domain separator. When injecting a fault such that the domain separator is reused, two values, such as s and e, are identical. Then, one can, e.g., trivially solve the equation $b = as + e = (a + 1)s$ for the key s.

Valencia et al. [VOGR18] performed a more general study of the susceptibility of LPR-like encryption schemes against fault attacks. They show attacks that target decryption and can recover the secret key. However, their attacks are applicable only to CPA-secure systems, such as plain LPR. Their techniques cannot be applied to, e.g., Kyber and NewHope, as the FO transform would detect all their injected faults, and the resulting pseudorandom key is of no use in a differential fault attack. Also, their attacks require faulting multiple decryptions that use the same key. This must be avoided in systems

providing only CPA security, as they could otherwise be easily broken with a chosen-ciphertext attack.

Thus, to the best of our knowledge, the susceptibility of FO-secured decapsulation has thus far not been properly investigated. We now change this in the following by presenting a new fault attack on broad range of lattice-based KEMs.

3 Generic Attack Description

In this section, we describe the underlying ideas and techniques of our attack. First, we describe the attacker's capabilities and discuss some trivial fault attacks, which we argue might either not be very realistic to mount or already obvious. Then, in Section 3.1, we observe that the decryption noise linearly depends on the secret polynomials e and s. In Section 3.2, we show that by injecting skipping faults in the decoder and then observing if the decapsulation still returns the correct shared secret (ineffective fault), one can restrict the possible values of the encryption noise and hence learn information on the long-term secrets. After repeating the above for many decapsulations, the attacker can create a linear system of inequalities, which we solve using a statistical method described in Section 3.3. Finally, we describe the efficient implementation of said statistical technique in Section 3.4.

The described combination of techniques makes our attack related to ineffective fault attacks [Cla07], statistical fault attacks [FJLT13, DEK+16], and their combination called statistical ineffective fault attacks [DEK+18, DEG+18].

Attacker model and steps. We assume that the adversary has the following capabilities and performs the now described basic steps. The attacker is in possession of the public key and can thus execute an arbitrary number of encapsulations. He gathers and stores not only the returned ciphertexts and shared secrets but also some intermediates computed during encapsulations, such as the sampled noise polynomials and the encrypted message. The attacker can filter the recorded encapsulation results according to some values known to him, e.g., the value of certain bits of the embedded encrypted message. However, he does not alter the ciphertexts, as the FO would always detect these modifications.

The adversary can then send a selected ciphertext to the target device, which holds the private key. He injects a specific skipping fault in PKE.Decrypt inside the decapsulation (cf. Figure 2), which is the only operation involving the private key s. Note, however, that the adversary cannot directly observe the decryption output μ'. Instead, the attacker can only infer if the device still derived the correct shared secret (the fault was *ineffective*) or if the fault attack caused an incorrect decryption result (the fault was *effective*). This can be done by, e.g., decrypting a response from the target using the shared key returned by the corresponding encapsulation. If the resulting plaintext conforms to the used protocol, then the targeted device still computed the correct shared secret. If, however, the plaintext appears to be random, then the FO detected the fault and thus returned a random shared secret.

Simple attacks. Before describing our attack, we now discuss some very basic fault attacks against FO-secured KEMs and argue why these attacks might be impractical or already obvious.

First, an attacker might want to skip the equality check during decapsulation, i.e., whether $c == c'$. This would then re-enable chosen-ciphertext attacks, such as shown in, e.g., [BGRR19]. However, fault injections might fail, and such attacks are typically not designed with uncertainties in mind. Also, we argue that the need to protect this obvious faulting target was already shown. In fact, skipping this check was previously proposed by Valencia et al. [VOGR18] in the context of their fault attacks. The need to protect the check was also noted by Oder et al. [OSPG18] and by Bauer et al. [BGRR19].

Another possible attack venue is exploiting ineffective faults in the stuck-at model [Cla07]. When an attacker can reliably fault an intermediate to a known value, e.g., to zero (stuck-at-zero), and observes that the faulted device still returns the correct shared secret, he can deduce that the intermediate was already zero. Such a fault-enabled probing capability allows trivial key recovery when targeting intermediates having, e.g., a linear dependency on the coefficients of s. However, such stuck-at faults are not trivial to achieve in practice, especially on 32-bit devices. Also, due to the large range of possible values, e.g., Kyber's modulus is 3329, it might be unlikely that the targeted intermediate takes on the injected value. When also considering the large key sizes, a vast number of highly reliable fault injections might be needed for a successful attack.

3.1 On the Linearity of the Decryption Noise

Instead of using one of the above approaches, our attack exploits the key dependency of the encryption noise. Recall from Section 2.1 that $m' = v - us = er - e_1 s + e_2 + m\lfloor q/2 \rfloor$, where we call $d = er - e_1 s + e_2$ the encryption noise. As noted above, we assume that the adversary (honestly) generated the ciphertexts. Hence, he knows all intermediate values computed during encapsulation, including e_1, e_2, and m. Now observe that the noisy plaintext m' and thus also the decryption noise d are linear combinations of known values and the secrets e and s.

Also, note that all components of d have small coefficients. This ensures that d is always in the range of $[-q/4, q/4]$; else decryption errors would occur.[4] Thus, one can essentially ignore the modular reductions and interpret the equation of d to be in $\mathcal{R} = \mathbb{Z}[x]/(x^n + 1)$.

If an adversary somehow recovers the entire noise d (or m') for two decryptions, then he can trivially solve the resulting system of linear equations for the unknowns e and s. A similar statement can be made for the much more realistic case where the attacker can only recover a single coefficient of d, but for many decryption queries. Here we can use the fact that polynomial multiplication in $\mathcal{R}_q$ can be rewritten into a matrix-vector product. That is, for $c = ab$, we can also write $c = Ab$, where, with a slight abuse of notation, we equate polynomials with their coefficient vectors. The i-th column of A can be generated by computing $a \cdot x^i$ in $\mathcal{R}_q$. This matrix-vector representation now allows to extract the computation of each individual coefficient of c. Concretely, we write $c[i] = \langle a^{(i)}, b \rangle$, where $a^{(i)}$ is the i-th row of A and $\langle \cdot, \cdot \rangle$ denotes an inner product. For the reduction polynomial $x^n + 1$ used in LPR, this equates to $c[i] = \sum_{0 \le m \le i} a[i-m] \cdot b[m] - \sum_{i < m < n} a[i+n-m] \cdot b[m]$. The above can easily be extended to scalar products of vectors of polynomials, as used by Kyber.

When an attacker now recovers $d[i]$ (or $m'[i]$, which allows him to compute $d[i]$), he can write $d[i] - e_2[i] = \langle r^{(i)}, e \rangle - \langle e_1^{(i)}, s \rangle$, i.e., a linear equation with $2n$ unknowns (the coefficients of e and s). Gathering $2n$ of these equations again allows for key recovery using linear algebra. This technique of extracting single linear equations over many calls and then solving the entire system was already shown to be useful for side-channel attacks on lattice-based schemes [GBHLY16].

3.2 Fault Injection in Decoding

With a fault attack, it is not trivial to recover the true value of any $d[i]$. Thus, we instead aim at using a relatively simple fault to acquire a *hint* on its value. Concretely, we inject a skipping fault in the decoder and then use the observed decapsulation outcome, i.e., whether the injected fault was effective or ineffective, as an oracle for $d[i]$. We now explain this approach on the example of Kyber's decoding routine described in Section 2.3.

[4] Correctly speaking, decryption errors can, at least in theory, still occur. However, the parameter sets are chosen such that their probability is negligible for our purposes.

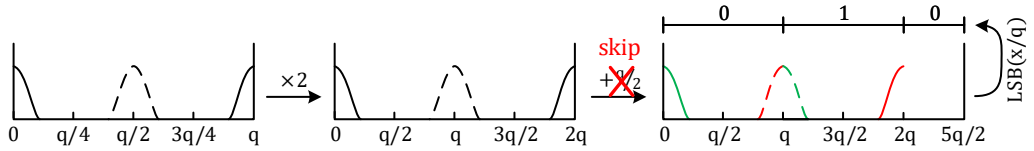


Figure 8: Visualization of Kyber's decoding routine if we skip the addition of $q/2$. Parts of the distribution shown in green are still correctly decoded, despite the fault injection (ineffective fault). Red parts are incorrectly decoded (effective fault).

We attack the decoding routine for one selected coefficient, e.g., for $m[i]$, and inject a fault such that the addition of $q/2$ (the second step in the visualization given in Figure 4) is skipped. Figure 8 now shows how this skipping fault influences the decoding process. Note that if the encryption noise $d \geq 0$, which corresponds to the right side of the distributions, then the correct value is still recovered. This is the case no matter the value of the encoded bit. Hence, the fault injection was *ineffective*. If, however, $d < 0$, then the faulted decoding returns an incorrect result; the fault is *effective*.

Thus, by injecting this skipping fault and then testing whether the attacked device still derives the correct key, we can infer if $d[i]$ is positive or negative. More formally, we have:

$$d[i] = \langle r^{(i)}, e\rangle - \langle e_1^{(i)}, s\rangle + e_2[i] \begin{cases} \geq 0 & \text{if fault was ineffective} \\ < 0 & \text{if fault was effective} \end{cases} \tag{2}$$

For now, we assume that our skipping fault has perfect reliability. We explain methods to deal with unreliable faults in Section 6.

3.3 Solving a System of Linear Inequalities

Each such equation gives a small amount of information on e and s. Thus, the adversary has to fault many decapsulations, where he can set up one equation per faulted execution. When using $r_j^{(i_j)}$ to denote the value of $r^{(i)}$ in the j-th decapsulation, and gathering a total of m equations, we can write:

$$\begin{pmatrix} r_1^{(i_1)} & -e_{1,1}^{(i_1)} \\ r_2^{(i_2)} & -e_{1,2}^{(i_2)} \\ & \vdots \\ r_m^{(i_m)} & -e_{1,m}^{(i_m)} \end{pmatrix} \begin{pmatrix} e \\ s \end{pmatrix} \begin{bmatrix} < \\ \geq \\ \vdots \\ < \end{bmatrix} \begin{pmatrix} -e_{2,1}[i_1] \\ -e_{2,2}[i_2] \\ \vdots \\ -e_{2,m}[i_m] \end{pmatrix} \tag{3}$$

The index i_j of the targeted coefficient can be different for each fault injection and thus each line in the system. We introduce dedicated symbols for the above system: we call the left matrix X, $y = (e||s)$, and the right-hand side of the equation z. The attacker now wants to find the single y satisfying all these constraints. We note that similar problems need to be solved for attacks exploiting decryption errors, e.g., [DGJ+19]. When encountering or enforcing such an error, one can infer that $|d| \geq q/4$ and use this information to construct a somewhat similar set of constraints.

Such a system of inequalities cannot be solved using straight-forward linear algebra. An approach that appears to be promising instead is linear programming (LP) since LP solvers already deal with such linear constraints. However, we found that the runtime of LP solvers grew very fast with the problem dimension, so much so that the required dimensions of more than 1000 unknowns appear to be out of reach. Also, LP solvers require hard constraints. However, fault injection might sometimes fail, which can then be

mistaken as an ineffective fault. A single resulting erroneous inequality might already be enough to eliminate the correct solution. Hence, a more resilient approach is needed.

We now describe a statistical approach that can deal with the large dimension and is (somewhat) error-tolerant. For each of the $2n$ secret coefficients, we store a vector of length $|\chi|$ containing the probabilities for all its values (all key guesses). These vectors are initialized with the probabilities given by the error distribution χ. We then use an iterated method to update said probabilities given all constraints.

Assume we want to infer information on $y[0] = e[0]$. For each of the $0 < k \leq m$ equations, we compute the probability distribution of $\sum_{j=1}^{2n-1} X[k,j]y[j]$, i.e., of the matrix-vector multiplication using all but the targeted secret coefficient. In the first iteration, the probabilities for the $y[j]$ are prescribed by χ, as stated above. In the next step, we enumerate all the guesses for $y[0]$. For each guess $y' \in \chi$, we compute $\Pr(y' + \sum_{j=1}^{2n-1} X[k,j]y[j] < z[0])$ if fault injection k was effective, or 1 minus this probability if the fault was ineffective. We do this for all m faults. Finally, we use the above probabilities to perform Bayesian updating of the probability of each key guess. Thus, in case of an effective fault, for each equation, we have

$$\Pr(y[0] = y') = \frac{\text{Prior}(y[0] = y')\Pr(y' + \sum_{j=1}^{2n-1} X[k,j]y[j] < z[0])}{\sum_{y^* \in \chi} \text{Prior}(y[0] = y^*)\Pr(y^* + \sum_{j=1}^{2n-1} X[k,j]y[j] < z[0])}$$

This process is also performed for all other secret coefficients in y. Importantly, we do not immediately use the updated probabilities. For instance, the update of $y[1]$ still uses the original priors of $y[0]$.

Performing this computation once for all coefficients, however, did not yield satisfactory results. For this reason, we repeat this entire process with the now updated probability vectors. After each such iteration, we pick the n most likely key values, which can then be plugged into the key relation $b = as + e$. The resulting linear system with n unknowns and n equations can be solved to recover the remaining n unknowns, given that the first n recovered coefficients are correct. We keep iterating the entire above process until either the correct key is recovered, or a defined maximum number of iterations–we set this threshold to 10–is reached.

We note that the described algorithm is related to the belief-propagation technique, which can be used for a plethora of inference tasks. It has seen some prior use in the context of side-channel analysis [VGS14], including attacks on lattice-based schemes [PPM17, PP19].

We also mention that the algorithm is not limited to working with inequalities, but can be adapted to exploit any information on $d[i]$. For instance, faults in other locations followed by the distinction between effective and ineffective faults might instead reveal the LSB of $d[i]$, or that $|d[i]|$ is within a certain range. Such information can also be incorporated.

3.4 Efficient Attack Implementation

The above is a relatively high-level description of our key-recovery algorithm. We now describe some steps required to make the recovery process practical.

Clustering. First, we decrease the dimension of the system by clustering multiple unknowns into single variables. The cluster sizes are chosen such that the number of key guesses can still be manageable given the runtime and memory constraints. In Kyber, for instance, the error polynomials are sampled from the centered binomial distribution over the range $[-2, 2]$, giving us 5 possible values per coefficient. We chose to cluster 4 coefficients, giving us $5^4 = 625$ key guesses per cluster. Additionally, after each iteration of our algorithm, we discard key guesses having very low probability and then merge clusters

such that their combined size is again enumerable. For ease of understanding, we do not consider clustering in the following explanations, but remember that it is still used.

Efficient summation of probability distributions. A major runtime hurdle in the above algorithm is the computation of the probability distribution of $\sum_{j=0\ldots 2n-1\setminus i} X[k,j]y[j]$ for all equations k and secret coefficients i. A naive summation is prohibitively expensive, which is why we use the following optimization.

When given two discrete random variables A, B and their respective distributions as a vector of probabilities, the probability distribution of $C = A + B$ can be computed by convolving the probability vectors of A and B. This convolution can be efficiently computed by utilizing the FFT and performing a pointwise multiplication of the transformed distributions. Thus, we compute $\mathrm{FFT}(\Pr(X[k,j]y[j]))$, where the probability vector first has to be zero-padded such that it can deal with the full expected range of d. Then, the pointwise product over the transformed vectors for all $j = 0 \ldots 2n-1\setminus i$ entries, followed by an inverse FFT, leads to the probability vector of the $\sum_{j=0\ldots 2n-1\setminus i} X[k,j]y[j]$.

Minimizing recomputations. The above needs to be computed for all coefficients i. Doing that via a simple loop over i would entail lots of recomputation of partial pointwise products. As an illustration, observe that products for two different i differ only by a single factor. We use dynamic programming to reuse such common factors. We store partial products in a binary tree, where the $2n$ leaves are initialized to $\mathrm{FFT}(\Pr(X[k,j]y[j]))$. We then move towards the root; for each node, we multiply the partial products of its children. We call these partial products the *upward distributions*.

After processing all nodes, we begin moving towards the leaves again and compute the *downward distributions*. In the layer below the root, the downward distributions are computed by multiplying the upward distributions of all respective siblings. For the next layers, the downward distribution is obtained by combining the downward distribution of the parent with the upward distribution of the siblings. Thus, for each node, the downward distribution describes the sum of all leaves that are *not* below the node. After fully traversing the tree in this manner, the required distributions are the downward messages of the leaves.

By using this method, the runtime complexity can be reduced from $\mathcal{O}(n^2)$ (simple loop over i) to $\mathcal{O}(n)$, at the cost of higher memory requirements.

4 Application to Kyber and NewHope

The previous section gave a somewhat generic description and targeted a combination of plain LPR with the FO. We now describe the steps needed to apply our attack on concrete schemes and their decoder implementations. We start with Kyber in Section 4.1, then discuss how the attack can be applied on a masked Kyber implementation in Section 4.2, and finally, in Section 4.3, show that other LPR-like schemes, concretely NewHope, can also be susceptible.

4.1 Kyber

The previous section explained the generic attack on Kyber's decoder but still targeted plain LPR. Some additional steps are needed to actually apply our attack to Kyber.

Recall that Kyber compresses both ciphertext components u and v via rounding, i.e., by, in simplified words, dropping the low order bits (cf. Section 2.3). This compression needs to be accounted for when setting up our system of inequalities. We write $u' = u + \Delta u$ and $v' = v + \Delta v$, where u, v are the uncompressed terms, u' and v' is the compressed ciphertext, and $\Delta u, \Delta v$ denotes the additive rounding error. We note that in our attack

scneario, the adversary performs the encapsulation and thus knows the uncompressed value and the rounding error.

When now performing back substitution in $v' - u's = (v + \Delta v) - (u + \Delta u)s$, similar to Equation (1), we arrive at:

$$m' = m \cdot \left\lfloor \frac{q}{2} \right\rfloor + er + e_2 + \Delta v - (e_1 + \Delta u)s$$

and thus have $d = er - (e_1 + \Delta u)s + e_2 + \Delta v$. The value of Δv can be somewhat large. Since our fault attack probes the sign of d, some inequalities will thus be fulfilled by (almost) all values of e and s. We thus filter for small $|e_2 + \Delta v|$, i.e., we only send those ciphertexts to the target where during encapsulation, this value was observed to be within some bound. Concretely, we only use the ciphertext of encapsulations where $|e_2 + \Delta v| \leq 10$. All other steps of the attack are performed as described in the previous section.

4.2 Kyber using the Masked Decoder

While masking is a powerful countermeasure against various side-channel attacks, it is, at least in general, not effective in protecting against faults. However, our attack probes the value of the (implicit) intermediate d, which only appears in masked form in a protected implementation. Hence, it might appear that side-channel protections, which are required anyway in a vulnerable environment, already thwart our attack.

We now show that this is not the case. While we do not claim that all masked implementations are susceptible, we demonstrate that masking as such does not hamper our attack. We do so by adapting the attack to the masked decoder of Oder et al. [OSPG18], which we introduced in Section 2.5.

Recall that the first step of this decoder is to subtract $q/4$ (mod q) and that the input to the decoder is arithmetically shared over $\mathbb{Z}_q$. As this subtraction is a linear operation in $\mathbb{Z}_q$, it suffices to perform it on just one of the two shares. The effects of skipping this subtraction are shown in Figure 9. The transformation to arithmetic shares mod 2^{bits} now splits the distribution for a 1-bit into two parts. The final subtraction of $q/2$ shifts the distributions such that no decoding error occurs when $d \geq 0$ (marked in green), but the wrong value is returned when $d < 0$ (marked in red). Thus, we observe the exact same effect as in the unmasked case. Hence, the next attack steps are identical.

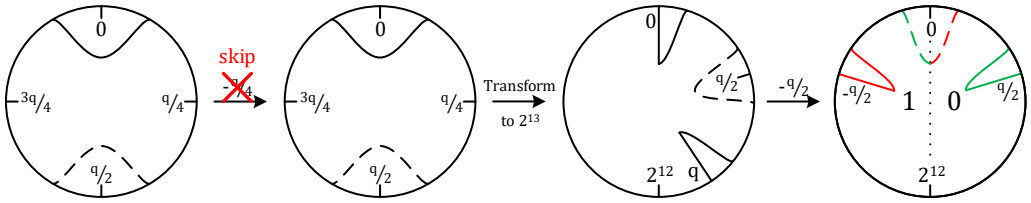


Figure 9: Visualization of the effect of skipping the first subtraction in the masked decoder of Oder et al. [OSPG18]

4.3 NewHope

Recall from Section 2.4 that NewHope encodes each message bit onto multiple coefficients. For decoding, first the FLIPABS function is run on all n coefficients of m'. Then, the 2 (for $n = 512$) coefficients encoding the same message bit are added. Finally, after subtracting $q/2$, the sign bit is returned.

We found that skipping a particular instruction in one call to the FLIPABS routine again allows us to learn the sign of d. Concretely, we skip the XOR with the bitmask m (line 8 of Figure 6). We now discuss the exact effect of this fault via a case study.

If the input $x \geq q/2$, i.e., x is on the right half of Figure 1b, then r is positive and $m = 0$. XORing a value of 0 has already no effect. Thus, the instruction skip is always ineffective, and the correct value is still returned.

If $x < q/2$, then $r < 0$ and $m =$ `0xffff`, which equals -1 for the `int16_t` data type. Skipping the XOR thus results in returning $x - q/2 - 1$ instead of the desired value $|x - q/2|$. This incorrect value is now added to the outcome of FLIPABS of the second coefficient encoding the same bit. We dub x_1 the result of the faulted FLIPABS call and x_2 the returned value of the undisturbed call, where x_2 follows the distribution shown in the right half of Figure 5. If the encoded message bit is 0, then we have $x_1 \in [-q/2 - 1, q/4 - 1]$ and $x_2 \in [q/4, q/2)$. The sum of these values is in $[-q/4 - 1, q/4 - 1)$, which is always smaller than $q/2$ and will thus incorrectly decode to a 1. If the original message bit is 1, then $x_1 \in [-q/4, -1]$ and $x_2 \in [0, q/4]$, which leads to a sum in $[-q/4, q/4)$. This will again be decoded to a 1, which is the correct value in this case.

For a message bit with value 0, $d \geq 0$ will thus lead to an ineffective fault, whereas $d < 0$ leads to corrupted result. Hence, we again set up a system of linear inequalities and solve for the key. For a message bit with value 1, the injected fault is always ineffective; the returned value is correct regardless of the value of d. We thus only send ciphertexts to the target where the attacked coefficient encodes a 0. The attacker cannot directly control the encrypted message, as it is generated by hashing a seed. However, the attacker can just discard seeds that lead to a 1 in the targeted position.

Compression. Unlike Kyber, NewHope only compresses the second ciphertext component v. We again filter for ciphertexts where $|\Delta v + e_2|$ is small.

5 Evaluation

In this section, we put the previous descriptions into practice and evaluate the performance of our attack. Using fault simulations, we analyze how many faulted decapsulations are needed for key recovery. We also investigate the computational-resource requirements for solving the system of inequalities with our algorithm.

Implementation. We implemented the statistical solving approach described in Section 3.3, including all optimizations mentioned in Section 3.4, in Matlab. All source code is available at https://github.com/latticekemfaults/latticekemfaults/.

For all evaluations presented in this section, we used fault simulations. That is, we modified the decapsulation of the Kyber and NewHope reference implementations such that the desired skipping fault is done in software. Thus, these evaluations assume perfect faulting reliability. However, we analyze a scenario with unreliable faults in Section 6. We analyzed all proposed Kyber parameter sets (Kyber512, Kyber768, and Kyber1024), and the smaller NewHope parametrization (NewHope512).

5.1 Number of Fault Injections

Each injected fault allows us to set up a linear inequality that carries a small amount of information on the long-term secret key. We now analyze how many such inequalities and thus fault injections are needed for full key recovery.

We performed a sweep over the number of inequalities for each mentioned parameter set. For each analyzed quantity, we perform 20 experiments and determine the success rate. The outcome for Kyber is shown in Figure 10. For the smallest parameter set, namely Kyber512, one needs approximately 6,500 fault injections to achieve a success rate above 90%. As to be expected, this number grows significantly for Kyber768 and Kyber1024. To achieve a similar success rate, we need 9,500 and 13,000 faults, respectively.

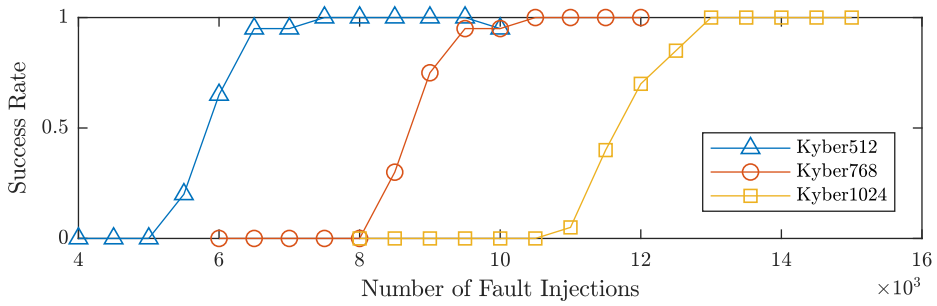


Figure 10: Attack success rate for all 3 Kyber parametrizations as function of faulted decapsulations

The result for NewHope512 is shown in Figure 11. Compared to Kyber512, attacking NewHope512 requires a much larger number of faults, despite being in the same NIST security category. We can attribute this, at least in part, to the larger key space. In Kyber, key coefficients are drawn from a centered binomial distribution over $[-2, 2]$, but in NewHope, they are sampled from $[-8, 8]$. We were not able to attack NewHope1024, potentially due to these reasons.

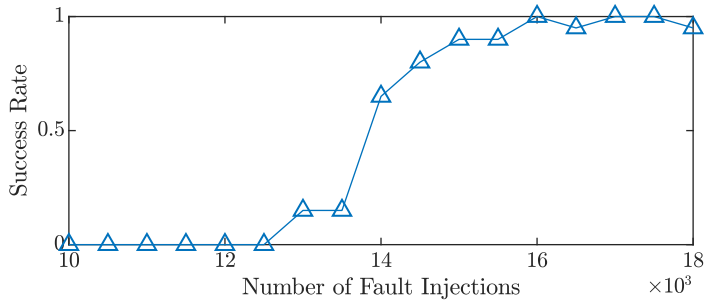


Figure 11: Attack success rate for NewHope512 as function of faulted decapsulations

5.2 Resource Requirements

Key recovery requires solving a very large system with at least 1024 unknowns and around 10,000 inequalities. Additionally, the algorithm does not assign a single value to the unknowns but instead needs to keep track of their entire probability distribution. Thus, the resource use of the solving algorithm is non-negligible.

The runtime and memory consumption depends on the size of the system, i.e., on both the number of unknowns and on the number of inequalities. The former is prescribed by the targeted parameter set. In Kyber, we have $2kn$ unknowns, which translates to 1024 for Kyber512. Successful attacks required at least 5,500 inequalities, but we used up to 18,000 during evaluations.

We give some exemplary resource requirements in Table 1. For each parameter set, we measured the runtime and memory requirements for one selected number of faults. We always picked the lowest number where the success rate, as per the analysis shown in Section 5.1, is at least 90%. All these measurements were done using 8 cores of a Xeon E5-4669 v4 2.2 GHz. The average runtime ranges between 3 and 20 minutes. Memory might be more of a limiting factor, as we required up to 79 GB of RAM. However, we did not particularly optimize our implementation in this regard, as our testing machine did still provide more than enough memory.

Table 1: Resource requirements for running the attack for one selected number of faults per parameter set. All measurements used 8 cores of a Xeon E5-4669 v4 2.2 GHz.

Parameter Set	Kyber512	Kyber768	Kyber1024	NewHope512
Faults	6,500	9,500	13,000	15,000
Avg. Runtime [min]	3	7	12	20
Max. RAM [GB]	24	40	61	79

6 Experimental Verification on an M4

The above experiments use simulations to show that a single skipping fault per decapsulation can indeed be enough for key recovery. To demonstrate that the attack is also practical, we successfully attacked Kyber512 running on an ARM-based microcontroller. This section describes the experiment and discusses its outcome.

6.1 Setup

For our experiment, we targeted an STM32F405 microcontroller featuring an ARM Cortex M4 core. The Cortex M4, and the STM32F40X series in particular, is the de-facto standard platform for evaluating embedded software implementations of schemes running in NIST's PQC process.

Our target device is mounted on the ChipWhisperer UFO board [Newb], which allows for relatively simple fault injection. Concretely, we use a ChipWhisperer Lite board [Newa] to generate the 24 MHz base clock, which the target device then directly uses as its core clock. We added a trigger signal to mark the beginning of the decoding process. We then inject a clock glitch such that a single chosen instruction is skipped, as described in the previous section. Determining the exact glitching parameters and the timing of the targeted instruction was done using sweeps over the glitching parameter space on an extracted implementation of the decoder.

We run the unprotected and in large parts assembly-optimized Kyber512-90s implementation included in the PQM4 library [KRSS]; the C portion of the code is compiled using the `O3` optimization level. The 90s version of Kyber replaces calls to Keccak with SHA2 and AES. We use this version simply because it runs slightly faster on our target. Apart from this, the choice of the used symmetric primitives does not affect our attack. We note that decoding is not manually optimized and instead uses the C code from the Kyber reference implementation. Still, as our attack requires only a single instruction skip, we do not expect worse attack performance when using an assembly-optimized routine.[5]

6.2 Attack Steps

We run encapsulation on a PC, which received the public key directly from the device. The PC can perform an arbitrary number of encapsulations and then decide which out of the generated ciphertexts to send to the device. As stated in Section 4.1, we only use the outcome of encapsulations where $|e_2 + \Delta v| \leq 10$, all other ciphertexts are discarded.

After sending such a selected ciphertext to the target device, it runs decapsulation, during which the described skipping fault is injected. For the sake of simplicity, the device directly replies with the computed shared secret key in clear. Thus, the attacker can directly test whether the received key matches the output of encapsulation (fault was ineffective) or if they differ (fault was effective). We note that in a real-world attack,

[5] We note that the ARMv7-M instruction set offers an addition with an integrated shift of one of the operands. This instruction could allow merging the bitshift and the addition of $q/2$. However, this instruction is only available for register-register addition. The compiler stores the constant $q/2$ as an immediate value, thereby disallowing the use of this merged instruction.

the adversary does not directly receive the shared key, but can, e.g., decrypt follow-up messages with the shared secret and test whether they adhere to the used protocol (correct shared key) or if they appear to be random garbage (incorrect shared key).

Dealing with unreliable injections. The described attack needs to distinguish between effective and ineffective fault injections. However, *failed fault injections*, i.e., cases where no instruction is actually skipped, cannot be distinguished from ineffective faults by just observing the shared secret. With our inexpensive setup, we were not able to achieve a high enough reliability for fault injection. Without further filtering, the attack failed due to the large number of incorrect inequalities. For this reason, we only use data from decapsulations returning an incorrect shared secret. In these cases, we can be certain that the fault injection worked and that the fault was effective. As we expect that about half the injected faults are ineffective, we thus have to discard the data from at least half of the faulted decapsulation queries.

Despite this filtering, some of the generated inequalities turned out to be incorrect. This can be the case when the injected clock glitch has some other (unknown) effect, such as general data corruption or skipping a different instruction. We call this scenario *unintended faults*. When assuming that the decoder returns a random bit in such a case, then unintended faults will be misclassified in about 50% of all injections. This percentage likely differs for a real setup, as the outcome of unintended faults can still show a bias. As it turns out, the statistical solution approach presented in Section 3.3 can deal with some incorrect inequalities without further adaptations. We discuss potential methods to achieve a higher robustness against unintended faults in Section 7.

6.3 Results

We ran 6 key recovery experiments, each one using a different key. For each experiment, we gathered 10,000 linear inequalities. With perfect faulting reliability, this would also be the number of required fault injections. As we filter for effective faults, we need to double that number. However, with our inexpensive setup, we did not achieve a very high faulting reliability. Some injections lead to crashing the device, while many others resulted in the correct shared key. As explained above, we need to discard such trials.

In 5 out of the 6 experiments, approximately 17% of the faulted decapsulations were exploitable and allowed the extraction of a linear inequality,[6] for the sixth run this number dipped to 8%. This corresponds to faulting roughly 60,000 and 125,000 decapsulations and required about 8.5 hours and 16 hours, respectively.

When feeding the gathered system of inequalities into our key-recovery algorithm, the correct private key was always successfully recovered. After plugging the recovered private key back into the inequalities, we found that between 0.4% and 1% of them were incorrect, presumably due to unintended faults. Still, this shows that the recovery algorithm is somewhat resilient against such errors.

As our goal was to demonstrate the feasibility of the attack, and as instruction skips are a well-established fault model and have been shown to work in the past, we did not put further effort into improving the above numbers. Still, we note that a determined adversary using more sophisticated faulting equipment can likely cut down the number of faulted decapsulations, ideally to the fault quantities found using simulations, and thereby drastically reduce the attack time.

[6] We can assume that roughly 50% of all injections should be classified as effective. We only see 17%; the remaining 33% are thus misclassified as ineffective. Thus, roughly 40% of all inequalities derived from injections classified as ineffective would be incorrect, thereby justifying the need to filter.

Robustness against erroneous inequalities. To get a better grasp on the robustness of our approach, we ran additional simulations with the above settings (Kyber512, 10,000 faults) and artificially introduced errors in the inequalities. With an error rate of 0.5%, the success rate is roughly 70%. When 1% of the inequalities are incorrect, the success rate drops into single-digit percentages. Since all practical experiments succeeded, we believe that the occurred errors still have some bias. We describe a method to deal with higher error rates in the following section.

7 Countermeasures and Future Work

While the FO inherently prevents many kinds of fault attacks, the presented method clearly shows that practical attacks are still possible. Our attack requires a single instruction skip per decapsulation and also shows resilience against erroneous fault injections. Thus, countermeasures are still needed. We now discuss several options and then conclude with possible future work.

7.1 Countermeasures

Protocol-level countermeasures. Our attack requires that the targeted device decapsulates any attacker-generated ciphertexts. Thus, if either the public key is only known to trusted parties or the target only runs decapsulation for ciphertexts signed by such a trusted party, the attack is prohibited. However, this might not be possible or practical in many applications.

Redundancy. Arguably the most popular method to protect against faults is to introduce redundancy and use it for error detection. The FO provides such redundancy through the involved re-encryption, but as shown, it does not prevent the attack.

The simplest form of redundancy is course-grained double execution, i.e., to run decapsulation twice and only return the shared secret if both results are equal. However, our attack does not require knowing the (faulty) shared secret; we only need to detect whether the injected fault changed the outcome. This is still possible with double execution in place, and possibly even easier if the device reacts differently upon detecting a fault.

Still, double execution can also be applied on a finer granularity. By using double computation to validate the integrity of the intermediates *within* the decoder, some instances of our attack can be prevented. For the attack on Kyber, for instance, we skip the addition of a constant. This will always be detected, no matter if the fault would change the outcome of the decoder. However, in the attack on NewHope described in Section 4.3, we skip the XOR with the bitmask m. For a message bit 0 and $d \geq 0$, m has a value of 0. As the XOR does not change any values in this case, the skipping fault is not detected, meaning it is still possible to differentiate between effective and ineffective faults.

Some lattice-based KEMs apply an error-correction code to the plaintext m and then run a decoder on the recovered plaintext. While this is done to increase the resilience against decryption errors, this also severely impedes our attack. Determining if our attack can be adapted to this setting would need further study.

CFI. A generic countermeasure that will always detect our attack is (fine-grained) control-flow integrity. We rely on skipping faults. Thus, an implementation which can ensure that the sequence of executed instructions is correct will not be vulnerable.

Shuffling. Finally, one very effective and easy to implement countermeasure appears to be shuffling. For setting up a linear inequality, we need to know *which* coefficient index i we faulted. Since the decoder is called for each coefficient independently, the order in which

the coefficients are processed can easily be shuffled. While an attacker can still target the decoder and even differentiate between effective and ineffective faults, the uncertainty on i prevents deriving the inequality and thus hinders the attack. As an additional bonus, this shuffling countermeasure also hampers side-channel attacks.

7.2 Future Work

Improving the robustness. For scenarios where the algorithm fails due to unintended faults, it can be adapted to cope with the larger error rates. One can estimate the expected error rate and incorporate this probability in the Bayesian update step. While this will inevitably increase the number of required fault injections, it will make the approach more robust.

Adaptation to other schemes. While we concretely apply the attack on the reference implementations of Kyber and NewHope, our descriptions of Section 3 target FO-protected plain LPR. Thus, many more LPR-based schemes (and their implementations) might be vulnerable against similar attacks. We now discuss some of these schemes.

- Saber [DKRV19] is a third-round finalist and thus a direct competitor to Kyber. Unlike Kyber and NewHope, it bases its security on the Learning with Rounding problem (LWR). For, e.g., key generation, it samples a random s and then generates the public key $b = \lfloor as \rceil$. That is, it does not sample a random e, but instead generates an error by rounding the product. Still, the above can be rewritten as $b = as + e$, with $e = \lfloor as \rceil - as$. Another differentiating feature is its use of a power-of-two modulus instead of a prime. Still, as it is structurally very similar to Kyber, and as the error due to rounding can be made explicit, our attack might be applicable.
- Frodo [NAB+19], which advanced to the third round as an *alternate candidate*, uses an unstructured lattice and encodes multiple message bits into each coefficient. We did not study the susceptibility of this approach.
- NTRU Prime [BCLvV19], also an alternate candidate in the third round, includes a variant NTRU LPrime, which, as the name already implies, can also be seen as being based on LPR. Hence, this variant might also be susceptible.
- LAC [LLJ+19] and Round5 [GMZB+19], both second-round candidates in NIST's PQC process, employ error correction codes on the encrypted plaintext. As stated earlier, this will severely impede our attack. Still, we do not rule out that the attack can be adapted to this setting.

Incorporating lattice reduction. Our statistical algorithm returns probabilities for each coefficient of the secret key s and the error vector e. Thus far, we simply picked the n coefficients with the highest confidence and then solved the linear system $b = as + e$.

A small number of errors in the recovered coefficients can be corrected by enumerating likely error positions. Alternatively, one can recover less than n coefficients, plug these into $b = as + e$, and then perform key recovery using lattice basis reduction. This requires a hard classification, i.e., information on the confidence cannot be further incorporated.

Recently, Dachman-Soled et al. [DDGR20] showed how such probabilities (soft information) can be incorporated into a lattice-reduction approach. There, however, it is important that the probabilities are somewhat reliable. We found that if our algorithm cannot recover the key, then it latches onto the most likely values and boosts their probability close to 1. We suspect that this is due to a positive feedback loop, i.e., the probability vector of a coefficient influences itself after at least two iterations. A possible solution is to abort our algorithm after a lower number of iterations, i.e., before the positive feedback has too much of an adverse effect.

Acknowledgements

This work has been supported by the Austrian Research Promotion Agency (FFG) via the K-project DeSSnet, which is funded in the context of COMET – Competence Centers for Excellent Technologies by BMVIT, BMWFW, Styria and Carinthia, and via the project ESPRESSO, which is funded by the province of Styria and the Business Promotion Agencies of Styria and Carinthia. This work has also been supported by the German Federal Ministry of Education and Research (BMBF) under the project "PQC4MED" (16KIS1041), and by the German Federal Ministry for Economic Affairs and Energy (BMWi) under the WIPANO-project "PoQuID" (03TNK011B).

References

[ABD+19] Roberto Avanzi, Joppe Bos, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, John M. Schanck, Peter Schwabe, Gregor Seiler, and Damien Stehlé. CRYSTALS–Kyber: Algorithm specification and supporting documentation (version 2.0). Submission to the NIST Post-Quantum Cryptography Standardization Project [NIS16], 2019. https://pq-crystals.org/kyber.

[BCLvV19] Daniel J. Bernstein, Chitchanok Chuengsatiansup, Tanja Lange, and Christine van Vredendaal. NTRU Prime. Submission to the NIST Post-Quantum Cryptography Standardization Project [NIS16], 2019. https://ntruprime.cr.yp.to/.

[BGRR19] Aurélie Bauer, Henri Gilbert, Guénaël Renault, and Mélissa Rossi. Assessment of the key-reuse resilience of newhope. In *CT-RSA*, volume 11405 of *Lecture Notes in Computer Science*, pages 272–292. Springer, 2019.

[Cla07] Christophe Clavier. Secret external encodings do not prevent transient fault analysis. In *CHES*, volume 4727 of *Lecture Notes in Computer Science*, pages 181–194. Springer, 2007.

[DDGR20] Dana Dachman-Soled, Léo Ducas, Huijing Gong, and Mélissa Rossi. LWE with side information: Attacks and concrete security estimation. In *CRYPTO (2)*, volume 12171 of *Lecture Notes in Computer Science*, pages 329–358. Springer, 2020.

[DEG+18] Christoph Dobraunig, Maria Eichlseder, Hannes Groß, Stefan Mangard, Florian Mendel, and Robert Primas. Statistical ineffective fault attacks on masked AES with fault countermeasures. In *ASIACRYPT (2)*, volume 11273 of *Lecture Notes in Computer Science*, pages 315–342. Springer, 2018.

[DEK+16] Christoph Dobraunig, Maria Eichlseder, Thomas Korak, Victor Lomné, and Florian Mendel. Statistical fault attacks on nonce-based authenticated encryption schemes. In *ASIACRYPT (1)*, volume 10031 of *Lecture Notes in Computer Science*, pages 369–395, 2016.

[DEK+18] Christoph Dobraunig, Maria Eichlseder, Thomas Korak, Stefan Mangard, Florian Mendel, and Robert Primas. SIFA: exploiting ineffective fault inductions on symmetric cryptography. *IACR TCHES*, 2018(3):547–572, 2018.

[DGJ+19] Jan-Pieter D'Anvers, Qian Guo, Thomas Johansson, Alexander Nilsson, Frederik Vercauteren, and Ingrid Verbauwhede. Decryption failure attacks on IND-CCA secure lattice-based schemes. In *Public Key Cryptography (2)*, volume 11443 of *Lecture Notes in Computer Science*, pages 565–598. Springer, 2019.

[DKRV19] Jan-Pieter D'Anvers, Angshuman Karmakar, Sujoy Sinha Roy, and Frederik Vercauteren. SABER. Submission to the NIST Post-Quantum Cryptography Standardization Project [NIS16], 2019. https://www.esat.kuleuven.be/cosic/pqcrypto/saber/.

[FJLT13] Thomas Fuhr, Éliane Jaulmes, Victor Lomné, and Adrian Thillard. Fault attacks on AES with faulty ciphertexts only. In *FDTC*, pages 108–118. IEEE Computer Society, 2013.

[Flu16] Scott R. Fluhrer. Cryptanalysis of ring-lwe based key exchange with key share reuse. *IACR Cryptol. ePrint Arch.*, 2016:85, 2016.

[FO99] Eiichiro Fujisaki and Tatsuaki Okamoto. Secure integration of asymmetric and symmetric encryption schemes. In *CRYPTO*, volume 1666 of *Lecture Notes in Computer Science*, pages 537–554. Springer, 1999.

[GBHLY16] Leon Groot Bruinderink, Andreas Hülsing, Tanja Lange, and Yuval Yarom. Flush, gauss, and reload - A cache attack on the BLISS lattice-based signature scheme. In *CHES*, volume 9813 of *Lecture Notes in Computer Science*, pages 323–345. Springer, 2016.

[GMZB+19] Oscar Garcia-Morchon, Zhenfei Zhang, Sauvik Bhattacharya, Ronald Rietman, Ludo Tolhuizen, Jose-Luis Torre-Arce, Hayo Baan, Markku-Juhani O. Saarinen, Scott Fluhrer, Thijs Laarhoven, Rachel Player, Jung Hee Cheon, and Yongha Son. Round5. Submission to the NIST Post-Quantum Cryptography Standardization Project [NIS16], 2019. https://round5.org/.

[HHK17] Dennis Hofheinz, Kathrin Hövelmanns, and Eike Kiltz. A modular analysis of the fujisaki-okamoto transformation. In *TCC (1)*, volume 10677 of *Lecture Notes in Computer Science*, pages 341–371. Springer, 2017.

[Inf17] Infineon Technologies. Ready for tomorrow: Infineon demonstrates first post-quantum cryptography on a contactless security chip. Press Release, 2017.

[KRSS] Matthias J. Kannwischer, Joost Rijneveld, Peter Schwabe, and Ko Stoffelen. PQM4: Post-quantum crypto library for the ARM Cortex-M4. https://github.com/mupq/pqm4.

[Lan16] Adam Langley. CECPQ1 results. https://www.imperialviolet.org/2016/11/28/cecpq1.html, 2016.

[LLJ+19] Xianhui Lu, Yamin Liu, Dingding Jia, Haiyang Xue, Jingnan He, Zhenfei Zhang, Zhe Liu, Hao Yang, Bao Li, and Kunpeng Wang. LAC. Submission to the NIST Post-Quantum Cryptography Standardization Project [NIS16], 2019.

[LPR10] Vadim Lyubashevsky, Chris Peikert, and Oded Regev. On ideal lattices and learning with errors over rings. In *EUROCRYPT*, volume 6110 of *Lecture Notes in Computer Science*, pages 1–23. Springer, 2010.

[NAB+19] Michael Naehrig, Erdem Alkim, Joppe Bos, Léo Ducas, Karen Easterbrook, Brian LaMacchia, Patrick Longa, Ilya Mironov, Valeria Nikolaenko, Christopher Peikert, Ananth Raghunathan, and Douglas Stebila. FrodoKEM. Submission to the NIST Post-Quantum Cryptography Standardization Project [NIS16], 2019. https://frodokem.org/.

[Newa] NewAE Technology Inc. CW1173 ChipWhisperer-Lite. https://wiki.newae.com/CW1173_ChipWhisperer-Lite.

[Newb] NewAE Technology Inc. CW308 UFO target. https://wiki.newae.com/CW308_UFO_Target.

[NIS16] NIST Computer Security Division. Post-Quantum Cryptography Standardization, 2016. https://csrc.nist.gov/Projects/Post-Quantum-Cryptography.

[OSPG18] Tobias Oder, Tobias Schneider, Thomas Pöppelmann, and Tim Güneysu. Practical cca2-secure and masked ring-lwe implementation. *IACR TCHES*, 2018(1):142–174, 2018.

[PAA+19] Thomas Pöppelmann, Erdem Alkim, Roberto Avanzi, Joppe Bos, Léo Ducas, Antonio de la Piedra, Peter Schwabe, Douglas Stebila, Martin R. Albrecht, Emmanuela Orsini, Valery Osheter, Kenneth G. Paterson, Guy Peer, and Nigel P. Smart. NewHope. Submission to the NIST Post-Quantum Cryptography Standardization Project [NIS16], 2019. https://newhopecrypto.org/.

[PP19] Peter Pessl and Robert Primas. More practical single-trace attacks on the number theoretic transform. In *LATINCRYPT*, volume 11774 of *Lecture Notes in Computer Science*, pages 130–149. Springer, 2019.

[PPM17] Robert Primas, Peter Pessl, and Stefan Mangard. Single-trace side-channel attacks on masked lattice-based encryption. In *CHES*, volume 10529 of *Lecture Notes in Computer Science*, pages 513–533. Springer, 2017.

[Reg05] Oded Regev. On lattices, learning with errors, random linear codes, and cryptography. In *STOC*, pages 84–93. ACM, 2005.

[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 *COSADE*, volume 11421 of *Lecture Notes in Computer Science*, pages 232–250. Springer, 2019.

[TU16] Ehsan Ebrahimi Targhi and Dominique Unruh. Post-quantum security of the fujisaki-okamoto and OAEP transforms. In *TCC (B2)*, volume 9986 of *Lecture Notes in Computer Science*, pages 192–216, 2016.

[VGS14] Nicolas Veyrat-Charvillon, Benoît Gérard, and François-Xavier Standaert. Soft analytical side-channel attacks. In *ASIACRYPT (1)*, volume 8873 of *Lecture Notes in Computer Science*, pages 282–296. Springer, 2014.

[VOGR18] Felipe Valencia, Tobias Oder, Tim Güneysu, and Francesco Regazzoni. Exploring the vulnerability of R-LWE encryption to fault attacks. In *CS2@HiPEAC*, pages 7–12. ACM, 2018.

A Parameter Sets of Analyzed Schemes

Table 2: Kyber parameter sets (second round) [ABD+19]

Parameter Set	Kyber512	Kyber768	Kyber1024
Ring dimension n	256	256	256
Modulus q	3329	3329	3329
Noise parameter η	2	2	2
Module rank k	2	3	4
Ciphertext compression (d_u, d_v)	$(10, 3)$	$(10, 4)$	$(11, 4)$

Table 3: NewHope parameter sets [PAA+19]

Parameter Set	NewHope512	NewHope1024
Ring dimension n	512	1024
Modulus q	12289	12289
Noise parameter η	8	8
Ciphertext compression d_v	3	3