Fault-Enabled Chosen-Ciphertext Attacks on Kyber

Julius Hermelink[1,2(✉)], Peter Pessl[1], and Thomas Pöppelmann[1]

[1] Infineon Technologies AG, Munich, Germany
{peter.pessl,thomas.poeppelmann}@infineon.com
[2] Research Institute CODE, Universität der Bundeswehr München, Munich, Germany
julius.hermelink@unibw.de

Abstract. NIST's PQC standardization process is in the third round, and a first final choice between one of three remaining lattice-based key-encapsulation mechanisms is expected by the end of 2021. This makes studying the implementation-security aspect of the candidates a pressing matter. However, while the development of side-channel attacks and corresponding countermeasures has seen continuous interest, fault attacks are still a vastly underdeveloped field.

In fact, a first practical fault attack on lattice-based KEMs was demonstrated just very recently by Pessl and Prokop. However, while their attack can bypass some standard fault countermeasures, it may be defeated using shuffling, and their use of skipping faults makes it also highly implementation dependent. Thus, the vulnerability of implementations against fault attacks and the concrete need for countermeasures is still not well understood.

In this work, we shine light on this problem and demonstrate new attack paths. Concretely, we show that the combination of fault injections with chosen-ciphertext attacks is a significant threat to implementations and can bypass several countermeasures. We state an attack on Kyber which combines ciphertext manipulation–flipping a single bit of an otherwise valid ciphertext–with a fault that "corrects" the ciphertext again during decapsulation. By then using the Fujisaki-Okamoto transform as an oracle, i.e., observing whether or not decapsulation fails, we derive inequalities involving secret data, from which we may recover the private key. Our attack is not defeated by many standard countermeasures such as shuffling in time or Boolean masking, and the fault may be introduced over a large execution-time interval at several places. In addition, we improve a known recovery technique to efficiently and practically recover the secret key from a smaller number of inequalities compared to the previous method.

1 Introduction

The emerging threat of large-scale quantum computers to asymmetric cryptography drives the research on quantum-secure schemes. The standardization

A. Adhikari et al. (Eds.): INDOCRYPT 2021, LNCS 13143, pp. 311–334, 2021.
https://doi.org/10.1007/978-3-030-92518-5_15

process on Post-Quantum cryptography (PQC) of the National Institute of Standards and Technology (NIST) [Natb] is in the third round, and as the process slowly reaches its (preliminary) end, the topic of implementation security of PQC schemes is receiving increased attention. While for classical public-key cryptography, implementation security has seen decades of research and threats are relatively well understood, the situation is much less clear for quantum-secure algorithms.

In this regard, lattice-based key-encapsulation mechanisms (KEMs) are of particular interest. These schemes offer comparably small key- and ciphertext sizes and high speeds. Therefore, they are especially well-suited for embedded devices, which are highly susceptible to implementation attacks. Also, NIST expects to pick one of the three third-round lattice KEMs for standardization by the end of 2021, thereby making the study of side-channel and fault attacks and possible mitigations a pressing matter [Nata].

In fact, the side-channel aspect has already seen some analysis. Especially single-trace (or more generally, profiled) attacks appear to be a focal point of research, in, e.g., [PPM17, PP19, ACLZ20, PH16]. Another emerging topic is the combination of side-channel analysis with chosen-ciphertext attacks as shown in [RRCB20] and [HHP+21]. Almost all finalist KEMs in the NIST process use some variant of the Fujisaki-Okamoto (FO) transform [FO99, HHK17] to achieve CCA security. However, chosen ciphertexts are still highly useful when combined with side-channel leakage from the re-encryption step involved in this transform [RRCB20, RBRC20, GJN20, BDH+21]. There also exist works on side-channel secured implementations [OSPG18, BDH+21, HP21, RRVV15, RRdC+16].

Fault attacks against these systems, however, are a comparably underdeveloped topic. A potential reason is that the FO transform closes many attack paths through its re-encryption step and ciphertext-equality test. One already well documented attack option is to skip the equality test and thereby re-enabling chosen-ciphertext attacks [VOGR18, OSPG18, BGRR19, XIU+21]. Very recently, however, Pessl and Prokop [PP21] showed that other parts of the decapsulation can also be sensitive to fault injections. By skipping over certain instructions and then observing if the device still computes the correct shared secret (ineffective fault) or not (effective fault), they can gather information on the secret key. After faulting many decapsulation calls and accumulating said information, they can solve for the key. While their attack is practical, it does not come without caveats. For instance, instructions-skip attacks typically require a certain level of knowledge on the used implementation, and they presume that the attacker can find the selected instruction in the execution time. Also, since the shuffling countermeasure randomizes the time of execution, the specific attack by Pessl and Prokop might be already defeated by this relatively cheap technique. Thus, it is unknown whether such standard countermeasures might already suffice when used on an FO-transformed KEM. It is also unclear what algorithm steps and intermediates require protection, and if more general data corruption can also be used for attacks.

Our Contribution. In this work, we explore these open questions and show that fault-enabled CCA-attacks are a powerful attack tool capable of bypassing many standard fault countermeasures. We also demonstrate that in these scenarios public data (the ciphertext) needs to be secured against manipulation. To show this, we present such a fault-enabled attack on Kyber, but instead of faulting the equality test within the FO-transformation, we exploit it as an oracle. We flip a single selected bit in an otherwise honestly generated ciphertext. This ciphertext is sent to the device under attack. The device decrypts and then re-encrypts the message before comparing the re-encrypted ciphertext to the sent ciphertext. Anytime between the initial unpacking and the final comparison of the ciphertexts, we correct the induced flip on either the sent or the recomputed ciphertext using a fault and observe whether the following comparison fails. From this, linear inequalities involving the secret key can be derived, akin to the attack by Pessl and Prokop [PP21]. We further improve upon their key-recovery; our belief propagation based recovery technique significantly reduces the amount of memory required and recovers the secret key using far fewer faults.

Our fault can, e.g., be introduced in memory during virtually the entire decapsulation, or at more specific locations in registers during re-encryption. That is, our attack is flexible in its target choice. Securing the equality check of the FO-transform does not protect against our attack. Several countermeasures, such as shuffling (which protects against [PP21]) and also Boolean masking, can be bypassed. In addition, our fault is introduced in public data only, making fault profiling easier. We thereby stress the importance of not only securing certain operations but also to protect the integrity of both secret and public data over most of the execution time. In addition, we show that countermeasures such as first-order masking, as presented e.g. in [OSPG18], on CCA2-secured LWE-based schemes are not sufficient to protect against fault attacks. Our results in this regard are similar to and in line with the results of [BDH+21] but in a more general setting and not reliant on a faulty comparison. In addition, compared to their approach, we provide a drastically improved key-recovery algorithm.

While we demonstrate our attack on Kyber, related attacks likely apply to conceptually similar schemes, such as Saber [DKRV18], FrodoKEM [ABD+21], or NewHope [AAB+19].

Outline. We first, in Sect. 2, give a short overview over necessary preliminaries, focusing on the Kyber [BDK+18] algorithm for reference in the later sections. We also give a short introduction to the belief propagation algorithm and the prior work of [PP21]. In Sect. 3, we describe the basic idea of our attack, leading to a description of the practical implementation and simulation in Sect. 4 and then give and discuss the results of our simulations. In Sect. 5, we explore possible countermeasures against our attack.

2 Preliminaries

2.1 Kyber

Kyber [BDK+18] is an IND-CCA2-secure key exchange mechanism (KEM) and a finalist in the NIST standardization process [Natb]. Kyber relies on the hardness

of the Module Learning with Errors (MLWE) problem [LS15] and thus belongs to the field of lattice-based cryptography. Three parameter sets are currently specified: Kyber512, Kyber768, and Kyber1024. Kyber internally uses a CPA-secure public-key encryption scheme (PKE) which can be seen as a descendant of the LPR scheme [LPR13]. From the PKE, a CCA-secure KEM is derived by using a variant of the Fujisaki-Okamoto (FO) transform [FO99, HHK17]. We now describe (simplified) versions of the KEM and the internally used PKE.

Kyber PKE. Computations in Kyber take place in R_q and R_q^k, with $R_q = \mathbb{Z}_q[x]/(x^n + 1)$ with $q = 3329$, $n = 256$, and $k \in \{2, 3, 4\}$ (depending on the parameter set). $\mathsf{Sample}_{\mathsf{Uniform}}$ performs coefficient-wise sampling from a uniform distribution over $\mathbb{Z}_q$, $\mathsf{Sample}_{\mathsf{Binom},\eta}$ denotes coefficient-wise sampling from a centered binomial distribution defined over $\{-\eta, -\eta+1, \ldots, \eta\}$, where $\eta \in \{\eta_1, \eta_2\}$. Sampling is deterministic and depends on a seed which is incremented after each call. The functions compress and decompress are given by

$$\mathsf{compress}(x, d) = \left\lceil \frac{2^d}{q} \cdot x \right\rfloor \bmod 2^d$$
$$\mathsf{decompress}(x, d) = \left\lceil \frac{q}{2^d} \cdot x \right\rfloor$$

where d is either set to d_u or d_v. The concrete values of all parameters are given in Table 1. The functions Encode and Decode[1] interpret a bitstream as polynomial and vice-versa. Decode therefor multiplies each bit of a message m by $\frac{q}{2}$. $\lceil \cdot \rfloor$ denotes rounding to the nearest integer and mod q maps an integer x to an element $x' \in \{0, 1 \ldots, q-1\}$ such that $x \equiv x' \bmod q$. Elements a under the number theoretic transform (NTT) are denoted by $\hat{a} = \mathrm{NTT}(a)$, and the symbol $\circ$ denotes pointwise multiplication. Vectors and matrices, i.e. elements in R_q^k and $R_q^{(k \times k)}$, are denoted in bold lowercase and uppercase letters, respectively. For an element $\mathbf{a} \in R_q^k$, $\mathrm{NTT}(\mathbf{a})$ is defined as applying the NTT component-wise to its components.

Table 1. Kyber parameter sets

Parameter set	q	n	k	(η_1, η_2)	(d_u, d_v)
Kyber512	3329	256	2	(3, 2)	(10, 4)
Kyber768	3329	256	3	(2, 2)	(10, 4)
Kyber1024	3329	256	4	(2, 2)	(11, 5)

Key generation (Algorithm 1) uniformly samples a matrix $\mathbf{A}$ using a seed ρ and samples vectors $\mathbf{s}, \mathbf{e}$ from a binomial distribution using a seed σ. The public key is then calculated as $\mathbf{t} = \mathbf{A}\mathbf{s} + \mathbf{e}$, the secret key is $\mathbf{s}$.

[1] In earlier works, the names of Encode and Decode were sometimes switched. We use them according to Kyber's specification.

Encryption (Algorithm 2) reconstructs $\mathbf{A}$, samples $\mathbf{r}, \mathbf{e_1}, e_2$ from the binomial distribution, and computes $\mathbf{u} = \mathbf{A^T r} + \mathbf{e_1}$ and $v = \mathbf{t}^T\mathbf{r} + e_2 + \mathsf{Decode}(m)$. The compressed $\mathbf{u}$ and v are returned as ciphertext. The decryption, depicted in Algorithm 3, computes an approximate version (due to compression being ignored here) of

$$\begin{aligned} & v - \mathbf{s}^T\mathbf{u} \\ &= \mathbf{t}^T\mathbf{r} + e_2 + \mathsf{Decode}(m) - \mathbf{s}^T\mathbf{A}^T\mathbf{r} - \mathbf{s}^T\mathbf{e_1} \\ &= \mathbf{s}^T\mathbf{A}^T\mathbf{r} + \mathbf{e}^T\mathbf{r} + e_2 - \mathbf{s}^T\mathbf{A}^T\mathbf{r} - \mathbf{s}^T\mathbf{e_1} + \mathsf{Decode}(m) \\ &= \mathsf{Decode}(m) + \mathbf{e}^T\mathbf{r} + e_2 - \mathbf{s}^T\mathbf{e_1} \end{aligned} \tag{1}$$

and as $\mathbf{e}$, $\mathbf{r}$, e_2, $\mathbf{s}$, and $\mathbf{e_1}$ are sufficiently small (due to being sampled from a narrow binomial distribution), the above gives m when encoded[2].

Algorithm 1. PKE.KeyGen (simplified)

Input: Seeds ρ, σ
Output: Public key pk, secret key sk

1: $\hat{\mathbf{A}} \in R_q^{k\times k} \leftarrow \mathsf{Sample}_{\mathsf{Uniform}}(\rho)$ ▷ Generate uniform $\hat{\mathbf{A}}$ in NTT domain
2: $\mathbf{s}, \mathbf{e} \in R_q^k \leftarrow \mathsf{Sample}_{\mathsf{Binom},\eta_1}(\sigma)$ ▷ Sample from binomial distribution
3: $\hat{\mathbf{s}} \leftarrow \mathsf{NTT}(\mathbf{s})$ ▷ NTT for efficient multiplication
4: $\hat{\mathbf{t}} \leftarrow \hat{\mathbf{A}} \circ \hat{\mathbf{s}} + \mathsf{NTT}(\mathbf{e})$ ▷ $\mathbf{t} \coloneqq \mathbf{As} + \mathbf{e}$
5: **return** $(pk \coloneqq (\hat{\mathbf{t}}, \rho), sk \coloneqq \hat{s})$

Algorithm 2. PKE.Encrypt (simplified)

Input: Public key $pk = (\hat{\mathbf{t}}, \rho)$, message m, seed τ
Output: Ciphertext ct

1: $\hat{\mathbf{A}} \leftarrow \mathsf{Sample}_{\mathsf{Uniform}}(\rho) \in R_q^{k\times k}$
2: $\mathbf{r} \in R_q^k \leftarrow \mathsf{Sample}_{\mathsf{Binom},\eta_1}(\tau)$
3: $\mathbf{e}_1 \in R_q^k, e_2 \in R_q \leftarrow \mathsf{Sample}_{\mathsf{Binom},\eta_2}(\tau)$ ▷ Sample from binomial distribution
4: $\mathbf{u} \leftarrow \mathsf{NTT}^{-1}(\hat{\mathbf{A}}^T \circ \mathsf{NTT}(\mathbf{r})) + \mathbf{e}_1$ ▷ $\mathbf{u} = \mathbf{A}^T\mathbf{r} + \mathbf{e}_1$
5: $v \leftarrow \mathsf{NTT}^{-1}(\hat{\mathbf{t}}^T \circ \mathsf{NTT}(\mathbf{r})) + e_2 + \mathsf{Decode}(m)$ ▷ $v = \mathbf{t}^T\mathbf{r} + e_2 + \mathsf{Decode}(m)$
6: $\mathbf{c}_1, c_2 \leftarrow \mathsf{Compress}(\mathbf{u}, v)$ ▷ Lossy compression
7: **return** $c \coloneqq (\mathbf{c}_1, c_2)$

Kyber KEM. To transform the PKE to a KEM using the FO-transform, two distinct hash functions H and G are required. The key generation of the KEM (Algorithm 4) corresponds mostly to that of the PKE. In the encapsulation (Algorithm 5) the shared secret K is derived from the hash of a uniform message m, the public key and the ciphertext c under PKE.Encrypt of the message,

[2] The function $\mathsf{Encode}(\mathsf{Compress}(\cdot))$ is often called Decoder by previous works.

Algorithm 3. PKE.Decrypt (simplified)

Input: Secret key $sk = \hat{\mathbf{s}}$, ciphertext $c = (\mathbf{c}_1, c_2)$
Output: Message m
1: $\mathbf{u}, v \leftarrow \mathsf{Decompress}(\mathbf{c}_1, c_2)$ ▷ Decompress ciphertext
2: $m \leftarrow \mathsf{Encode}(\mathsf{Compress}(v - \mathsf{NTT}^{-1}(\hat{\mathbf{s}}^T \circ \mathsf{NTT}(\mathbf{u}))))$ ▷ Retrieve m
3: **return** m

with seed τ depending on the message and the public key. The decapsulation (Algorithm 6) decrypts the ciphertext using PKE.Decrypt and re-encrypts the message using the randomness retrieved from the message and the public-key hash $H(pk)$. If the re-encrypted ciphertext c' matches the received ciphertext c, the shared secret is returned, otherwise, the secret value z is used for an implicit rejection.

Algorithm 4. Kyber-KEM Key Generation (simplified)

Output: Public key $pk = pk_{pke}$, secret key $sk = (sk_{pke}||pk||H(pk)||z)$
1: $z, \rho, \sigma \leftarrow \mathsf{Sample}_{\mathsf{Uniform}}$
2: $pk_{pke}, sk_{pke} \leftarrow$ PKE.KeyGen(ρ, σ)
3: **return** $(pk \coloneqq pk_{pke}, sk \coloneqq (sk_{pke}||pk||H(pk)||z))$

Algorithm 5. Kyber-KEM Encapsulation (simplified)

Input: Public key $pk = (\hat{\mathbf{t}}, \rho)$
Output: Ciphertext c, shared key K
1: $m, \tau \leftarrow \mathsf{Sample}_{\mathsf{Uniform}}$ ▷ Uniformly sample a message and a seed
2: $(\bar{K}, \tau) \leftarrow \mathsf{G}(m||\mathsf{H}(pk))$
3: $c \leftarrow \mathsf{PKE.Encrypt}(pk, m, \tau)$ ▷ CPA encryption with seed τ
4: $K \leftarrow \mathsf{KDF}(\bar{K}||\mathsf{H}(c))$ ▷ Derive shared key from ct, m, and pk
5: **return** (c, K)

2.2 Belief Propagation

Since the belief propagation algorithm is an important part of our attack, we now give a brief introduction based on the description of MacKay [Mac03, Chapter 26]. For random variables $\{x_i\}_{i\in\{1,\dots,N\}} = \mathbf{x}$ on a set X with joint mass function

$$p(\mathbf{x}) = \prod_{k=1}^{K} f_k(\mathbf{x}_{I_k}), x \in X$$

Algorithm 6. Kyber-KEM Decapsulation (simplified)

Input: Secret key $sk = (\hat{\mathbf{s}}, pk, z)$, ciphertext $ct = (\mathbf{c}_1, c_2)$
Output: Shared key K

$m \leftarrow \mathsf{PKE.Decrypt}(sk, ct)$
$(\bar{K}, \tau) \leftarrow \mathsf{G}(m||\mathsf{H}(pk))$ ▷ Retrieve seed for re-encryption
$c' \leftarrow \mathsf{PKE.Encrypt}(pk, m, \tau)$ ▷ Re-encrypt
if $c = c'$ **then**
    $K \leftarrow \mathsf{KDF}(\bar{K}||\mathsf{H}(c))$ ▷ Derive shared key on successful re-encryption
else
    $K \leftarrow \mathsf{KDF}(z||\mathsf{H}(c))$ ▷ Implicit rejection on failure
return K

with $I_k \subseteq \{1, \ldots, N\}$ and f_k functions mapping $\mathbf{x}_{I_k} = \{x_i\}_{i \in I_k}$ to $[0, 1]$, belief propagation aims on efficiently computing all

$$Z_n(x) = \sum_{\mathbf{x}, \mathbf{x}_n = x} p(\mathbf{x})$$

which are proportional to the marginal distributions of x_n. Naïvely computing Z_n is often computationally infeasible. Belief propagation exploits the factorisation of p to significantly reduce the complexity of computing the marginals.

To compute the Z_n, and thereby the marginal distributions of x_n, for all n, the $x_1, \ldots, x_N$ are interpreted as *variable nodes*, which are connected to *factor nodes* given by $f_1, ..., f_K$, where the connections are described by the index sets $I_1, ..., I_K$. In each step, either variable nodes or factor nodes send messages to connected nodes. Messages at a variable node (with index) $i \in \{1, \ldots, N\}$ to factor node $j \in \{1, \ldots, K\}$ are computed as

$$m_{i,j}(x) \leftarrow \prod_{k \neq j} m'_{k,i}(x)$$

where $m'_{k,i}$ are the messages that were sent to the i-th node in the previous iteration. Messages from a factor node j to a variable node i are computed as

$$m_{j,i}(x) \leftarrow \sum_{\mathbf{x}, \mathbf{x}_i = x} f_j(\mathbf{x}) \prod_{k \neq i} m'_{k,j}(x).$$

Belief propagation consists of the repeated computation and passing of messages from variable to factor nodes and vice-versa. For an acyclic graph, the product of the messages at the n-th variable node converges against Z_n. Belief propagation on cyclic graphs is called *loopy belief propagation* and often gives useful approximations, even graphs might not converge or, in the context of key recovery, retrieve the correct result.

Belief propagation has proven to be a powerful tool for side-channel analysis of a wide variety of schemes [VGS14, PPM17, GRO18, PP19, KPP20, GGSB20], recently also in combination with chosen-ciphertext attacks [HHP+21].

In this work, we use belief propagation to find the most likely solution in a linear system of inequalities. This is similar to the usage of belief propagation for decoding, e.g., low density parity check (LDPC) codes [Mac03], where one aims at solving a system of (possibly erroneous) linear equations. In our case, every variable node has a so called *prior* distribution, which is set to the distribution the corresponding unknown variable was sampled from. In every iteration, either all variable nodes process messages and pass them to factor nodes or vice-versa. A *full iteration* is the combination of iterations from variable to factor nodes and vice-versa.

2.3 The Fault Attack of Pessl and Prokop

In [PP21], Pessl and Prokop introduced an instruction-skipping fault in the Decompress/Decode method of Algorithm 3. They thereby provided proof that the equality check of the FO-transform is not the only critical operation for fault attacks.

By introducing said fault and observing whether the re-encryption comparison in Algorithm 6 fails (by testing if the device still returns the correct shared secret K), they obtain a linear inequality involving the decryption error polynomial (c.f. Eq. (1)). In the j-th fault introduction, this error polynomial is given by

$$\mathbf{e}^T\mathbf{r}_j - \mathbf{s}^T(\mathbf{e_1}_j + \Delta\mathbf{u}_j) + e_{2j} + \Delta v_j \tag{2}$$

for $j \in \{0, ..., l-1\}$ where l is the total number of faults introduced, and where $\Delta\mathbf{u}_j$ and Δv_j denote the compression error introduced to $\mathbf{u}_j$ and v_j, respectively, by applying compression and decompression. Note that $\mathbf{r}$, $\mathbf{e_1}$, e_2, $\Delta\mathbf{u}$, Δv, as well as the true shared secret K are all known to the attacker, assuming he honestly performs encapsulation.

Denoting the t-th component of a vector of polynomials by $\mathbf{r}^{(t)}$, by writing out Eq. (2), we get that the i-th coefficient of the error term polynomial is given by

$$\begin{aligned}
&\sum_{t=0}^{k-1}\sum_{h=0}^{n}\sigma(h,i)\mathbf{e}_h^{(t)}\mathbf{r}_{j,\tau(h,i)}^{(t)} \\
+\ &\sum_{t=0}^{k-1}\sum_{h=0}^{n}\sigma(h,i)\mathbf{s}_h^{(t)}({\mathbf{e_1}}_{j,\tau(h,i)}^{(t)} + \Delta\mathbf{u}_{j,\tau(h,i)}^{(t)}) \\
+\ &e_{2j,i} + \Delta v_{j,i},
\end{aligned}$$

where $\tau(h,i) = i-h \bmod n$, and $\sigma(h,i)$ returning 1 if $i-h \geq 0$ and -1 otherwise. Using the notation $(\cdot)_i$ to encapsulate all involved sign flips and index shifts, we can restate the above using dot products:

$$\langle(\mathbf{r}_j)_i, \mathbf{e}\rangle + \langle(\mathbf{e_1}_{,j} + \Delta\mathbf{u}_j)_i, \mathbf{s}\rangle + e_{2j,i} + \Delta v_{j,i}$$

Assuming that i is constant over all injections, the inequalities, involving the error polynomial, may thus be written in matrix-vector form as $Ax \lessgtr -b$ where

$$A = \begin{pmatrix} (\mathbf{r}_0)_i & (\mathbf{e_1}_0 + \Delta\mathbf{u}_0)_i \\ (\mathbf{r}_1)_i & (\mathbf{e_1}_1 + \Delta\mathbf{u}_1)_i \\ \vdots & \\ (\mathbf{r}_{l-1})_i & (\mathbf{e_1}_{l-1} + \Delta\mathbf{u}_{l-1})_i \end{pmatrix},$$

$$x = \begin{pmatrix} \mathbf{e} \\ \mathbf{s} \end{pmatrix}, \text{ and } b = \begin{pmatrix} e_{2_{0,i}} + \Delta v_{0,i} \\ e_{2_{1,i}} + \Delta v_{1,i} \\ \dots \\ e_{2_{l-1,i}} + \Delta v_{l-1,i} \end{pmatrix}.$$

Note that these inequalities hold over $\mathbb{Z}$ as due to all polynomials involved being small, no reduction modulo q happens.

Each coefficient of each polynomial of $\mathbf{e}$ and $\mathbf{s}$ was sampled from a known binomial distribution with small support. To recover the key from the inequalities above, Pessl and Prokop initialize a $2n$ vector of probability distributions using said distribution. This vector is successively updated in each iteration according to the information given by the system of inequalities represented by A and b.

As previously mentioned, the attack is highly implementation specific and requires a high level of synchronisation, as one needs to skip over a very specific instruction. Also, the attack can likely be prevented by simple shuffling. Hence, the true potential of fault attacks is still unclear.

3 Enabling Chosen-Ciphertext Attacks with Faults

In this section, present an attack that improves on the mentioned weaknesses of [PP21]. Concretely, we explain how to use a fault to enable a chosen-ciphertext attack on Kyber. Instead of faulting the decoder, we assume an attacker to be able to introduce a single-bit fault. Sending a manipulated ciphertext and then correcting it back to a valid ciphertext after the decryption step using a fault yields inequalities, similar to [PP21]. This is due to the fact that the success of decapsulation implies that decrypting the manipulated ciphertext yields the original message used to create the valid ciphertext. This means, our attack consists of

1. manipulating the polynomial v (in Algorithm 2) of a valid ciphertext during encapsulation in Algorithm 5 by flipping a single bit in the compressed ciphertext,
2. sending the manipulated ciphertext to the device under attack,
3. correcting the ciphertext during decapsulation (Algorithm 6) using a one-bit fault,

4. observing whether a valid shared secret is established,
5. deriving inequalities from repeating the above steps (one inequality per fault),
6. and recovering the secret key from those inequalities.

As shown in Fig. 5, being able to introduce one-bit faults anywhere in the red phases enables our attack. Decompression and compression methods are depicted here, even though they belong to the decryption and the re-encryption, as they prevent an earlier manipulation of c'. The incoming ciphertext c is manipulated and either c or c' need to be "corrected" using a fault. That is, either c is faulted such that it matches the unaltered ciphertext, or c' is faulted such that it matches the manipulated ciphertext. Introducing a fault in c can be done by e.g. flipping a bit in RAM, while manipulation of c' would likely be done using a fault against a value in a register, as values generated during the re-encryption might not be stored RAM, or only for a shorter duration and possibly varying addresses. The decryption/re-encryption corresponds to lines 1 to 3 in Algorithm 6, the compression and decompression methods correspond to lines 5 and 1 of Algorithm 2 and Algorithm 3 which are called from lines 1 and 3 of Algorithm 6, respectively.

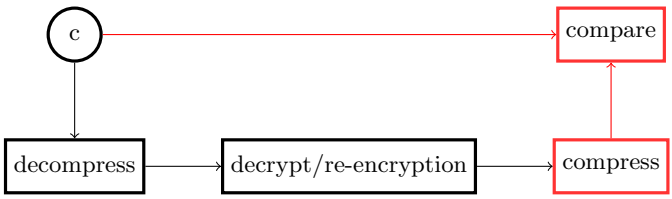

Fig. 1. Visualisation of the FO-transforms re-encryption check including decompression and compression. Being able to introduce one-bit faults anywhere in the red (light) phases enables our attack. (Color figure online)

3.1 Manipulating and Correcting the Ciphertext

The PKE ciphertext consists of the result of compressing a polynomial v (to get c_2) and of compressing a vector of polynomials $\mathbf{u}$ (to get c_1). The decrypt functions decompresses both c_1 and c_2, retrieves approximate versions of $\mathbf{u}$ and v, and computes and approximate version of $v - \mathbf{u}^T\mathbf{s}$, given by

$$rec = \mathbf{e}^T\mathbf{r} - \mathbf{s}^T(\mathbf{e_1} + \Delta\mathbf{u}) + e_2 + \Delta v + \text{Decode}(m)$$

where each coefficient is reduced to the range $\{0, ..., q-1\}$ and the Δ-terms denote the difference introduced by first compressing and then decompressing a (vector of) polynomial(s). The message is then recovered by mapping the coefficients of rec, $rec[i]$ for $i \in \{0, \ldots, n-1\}$, to a 0-bit if $rec[i]$ is closer to 0 or q than to

$q/2$ and to a 1-bit, otherwise; i.e. the function mapping a coefficient, reduced to $\{0, ..., q-1\}$, to a bit is given by

$$\phi : \{0, ..., q-1\} \to \{0, 1\}$$
$$a \mapsto \begin{cases} 0, & \text{if } \min(|a-q|, |a|) < q/4 \\ 1, & \text{else.} \end{cases}$$

For an honestly generated ciphertext, this yields the message m with high probability as the error polynomial d, given by

$$d = \mathbf{e}^T\mathbf{r} - \mathbf{s}^T(\mathbf{e_1} + \Delta\mathbf{u}) + e_2 + \Delta v,$$

is small.

Decoding of Manipulated Ciphertexts. By adding $q/4$ to $rec[i]$, we in some cases change the i-th bit m_i of the decoded message m. If m_i is 0, $rec[i]$ is "closer" to 0 mod q, i.e., 0 or q, than to $q/2$. In the first case ($rec[i]$ is close to 0), adding $q/4$ changes the decoding result to 1, as the result is now closer to $q/2$. In the second case ($rec[i]$ is close to q) $rec[i] + q/4$ will still be decoded to 0 (the result is now closer to 0 than to $q/2$). Analogously, if m_i is 1, then adding $q/4$ to $rec[i]$ changes the result of decoding to 0 if $rec[i] < q/2$.

Hence, observing if decoding $rec[i] + q/4$ still results in the same message m allows to retrieve information about the error polynomial d. By modularly mapping the coefficients of d to $\{-\lfloor \frac{q}{2} \rfloor, \ldots, \lfloor \frac{q}{2} \rfloor\}$ and ignoring rounding, this may be as expressed as

$$rec[i] + \frac{q}{4} \text{ decodes to } m_i \text{ if, and only if, } d[i] < 0.$$

We are using this property of ϕ to manipulate ciphertexts and introduce faults such that we may derive a system of inequalities involving the secret key.

Correct vs. Incorrect Message. Recall that during decapsulation (Algorithm 6), the randomness τ used for re-encryption is derived by hashing the message m with the hash of the public key. Hence, flipping just a single bit in m yields a completely random c'. However, if the correct m is still computed, then c' will be equal to the original (non manipulated) ciphertext and thus only differs in a single bit from the sent ciphertext.

Introducing and Correcting an Error. The above observations lead to the following method. In the encapsulation step, we first create a valid ciphertext $c = (v, \mathbf{u})$ where $v = \sum_{i=0}^{n-1} v_i x^i$. We then replace v by $v' = v + \frac{q}{4}x^i$ and compress it to obtain a manipulated ciphertext c', which is sent to the device under attack. The attacked device decompresses c', retrieves a message m', re-encrypts the message to a ciphertext c'' and compares c' against c''. Before the comparison, we introduce a fault, flipping a bit of c', such that $c' = c$. Thereby, we achieve that

the attacker performs the decryption on the manipulated ciphertext c' but effectively compares against the honestly generated ciphertext c. Observing whether a shared secret is established then tells us if the decryption of the manipulated ciphertext c' resulted in the original message m which was used to generate c. We note that this approach (manipulating a single coefficient of v and then testing if a decryption failure occurs) bares some resemblance to the side-channel attack of [BDH+21] on (flawed) algorithms for masked comparison [OSPG18, BPO+20].

Retrieving Information from Observing Encryption Failures. In the decryption step of the decapsulation routine, replacing c by the manipulated ciphertext c' results in the message being recovered from

$$v' - \mathbf{s}^T\mathbf{u} = v - \mathbf{s}^T\mathbf{u} + \left\lfloor \frac{q}{4} \right\rceil x^i = rec + \left\lfloor \frac{q}{4} \right\rceil x^i.$$

The manipulation therefore only affects the i-th message bit m_i and does not produce the same message and thus prevents a failed shared secret from being established if $d[i] > 0$ and $m_i = 0$ or $d[i] \geq 0$ and $m_i = 1$[3]. The probability of an error occurring not introduced by the manipulated ciphertext is those of a decryption failure and can be ignored for this attack.

Restricting to Single-Bit Differences. If c and c' differ by more than one bit, we do not use that ciphertext and re-try with new randomness. Otherwise, single-bit faults would not be sufficient. This may take a few tries, but as ciphertexts can be pre-computed and are not sent to the device, we do not regard this as a limitation. Note that allowing multi-bit differences would also result in slightly different inequalities as in these cases, compression changes the error added to v.

Fault Location and Profiling. A single-bit fault model at a specific time in execution is realistic but requires profiling or good understanding of the implementation of software as well as hardware [RSDT13, OGM17]. However, in our case, the fault may be introduced over a time interval spanning almost the whole execution time and on different intermediates and methods, e.g. against a value in memory or in a register, as depicted in Fig. 1. Note that the attacker knows the value of the bit to be flipped and may discard ciphertexts with an undesired value of the targeted bit. Thus, a bit flip can be achieved if the attacker is being able to either set or reset bits, which is a more realistic attacker assumption compared to straight-up flipping [RSDT13, OGM17]. If a Boolean-masked value is targeted, then setting/resetting a bit is ineffective with a probability of 50 %, even if the plain value is known. Still, if we observe a successful decapsulation (no decryption failure), then we can infer that the bit was actually flipped. As we cannot further distinguish the cause of a decapsulation failure (decryption failure or ineffective fault injection), we have to ignore injections yielding this result. Therefore, in this case, the number of required faults is approximately quadrupled.

[3] The difference in strictness arises from rounding to integers.

In addition, the observed outcome of the FO can be used for profiling, i.e., finding the correct faulting position in memory. The sent manipulated ciphertext will always be rejected unless properly corrected using a fault. This means that one can sweep over faulting positions and accept the one which leads to a correct decapsulation.[4] Even after finding a proper position, discarding inequalities resulting from observed decapsulation failures allows to filter out fault injections which did not produce the desired effect.

We finally note that the fault target (the ciphertext c) is public. Using, e.g., power analysis, one can find the point in time at which c is written into memory. This can aid in finding its address and physical location in RAM.

3.2 Obtaining Inequalities

To obtain inequalities in $\mathbf{e}$ and $\mathbf{s}$, we apply the procedure described in the previous section l times, where in the j-th step we obtain an inequality involving the i-th coefficient of the error polynomial, denoted d_j. As described before, in case of the decapsulation failing, we have $d_j[i] \geq 0$ and otherwise $d_j[i] \leq 0$[5]. The polynomial d_j is given by

$$\begin{aligned} d_j &= \psi(v_j) - \mathbf{s}^T\psi(\mathbf{u_j}) \\ &= (v + \Delta v) - \mathbf{s}^T\mathbf{u_j} - \mathbf{s}\boldsymbol{\Delta}\mathbf{u} \\ &= \mathbf{e}^T\mathbf{r}_j - \mathbf{s}^T(\mathbf{e_1}_j + \Delta\mathbf{u}_j) + e_{2j} + \Delta v_j + \mathrm{Decode}(m), \end{aligned}$$

where ψ denotes $\mathsf{Decompress}(\mathsf{Compress}(\cdot))$. This means our inequalities are of the form

$$(\mathbf{e}^T\mathbf{r}_j - \mathbf{s}^T(\mathbf{e_1}_j + \Delta\mathbf{u}_j))[i] \lessgtr (e_{2j} + \Delta v_j)[i],$$

where i is the coefficient we are manipulating/faulting, j is the index of the current step/fault, and all variables except for $\mathbf{e}$ and $\mathbf{s}$ are known. As the inequalities are clearly linear in $\mathbf{e}$ and $\mathbf{s}$, we may write those equations as $Ax \lessgtr -b$ where each row of the matrix A, together with the corresponding row of b and the information whether this row corresponds to a smaller or a greater sign, gives a check node as described in the next section. The construction of A and b is analogue to the construction in Sect. 2.3, x is the vector consisting of the entries of $\mathbf{e}$ and $\mathbf{s}$. In contrast to [PP21], we are not directly using the matrix structure. As described in the next section, each inequality, i.e. each row of A and b, are represented by a check node in a belief propagation graph.

[4] By first adding, e.g., only $q/8$ instead of $q/4$ to one coefficient of v, the chance that $m = m'$ and thus the probability of acceptance after a successful fault is drastically increased. This allows finding neighboring bits in memory more easily, which can then be used to find the actual targeted bit.

[5] By taking the i-th message bit into consideration, one may derive strict inequalities.

3.3 Recovering the Secret Key

To recover the secret key from inequalities, we use belief propagation. Our belief propagation graph consists of *variable nodes*, representing each unknown coefficient of $\mathbf{e}$ and $\mathbf{s}$, and factor nodes, which inspired by decoding algorithms we call *check nodes*, representing an inequality. Check nodes are connected to all variable nodes. In each iteration, the check nodes send messages to the variable nodes or vice-versa. A message represents a probability distribution over $\{-\eta_1, \ldots, \eta_1\}$ of a key coefficient, where we interpret the key as consisting of $\mathbf{e}$ and $\mathbf{s}$ and denote the key vector as x. In each step the messages are combined according to the inequalities represented by the respective check node, from which probability distributions for each key coefficient are derived. Figure 2 shows a simplified example with four unknown variables (represented by x_0, x_1, x_2, x_3) and five inequalities (represented by *Check 0*, ..., *Check 4*).

The variable nodes are initialized with the distribution $\mathbf{e}$ and $\mathbf{s}$ were sampled from in Algorithm 1, this is called the prior distribution and in the following denoted by *prior*. In the first step, the prior distributions, now called *messages*, are sent to the check nodes. The check nodes update the distributions according to the inequality they represent. For an unknown coefficient i, all other messages are combined according to the represented inequality and the resulting probability distribution is used to derive a distribution for the i-th coefficient. To be precise, for each input with index i every check node computes the distribution of the sum in the inequality, leaving out i. This means a check node with index j, corresponding to an inequality

$$\sum_i a_{ji} x_i \lessgtr b_j$$

receiving messages $m_0, \ldots, m_{2n-1}$ (m_i belongs the i-th key coefficient) computes distributions

$$D_i = \sum_{i \neq j} a_{ji} m_i \text{ for } i \in \{0 \ldots 2n\} \tag{3}$$

where addition corresponds to computing the distribution of the sum of the corresponding random variables, i.e. convolution of the m_i (see Sect. 4), and sends the messages

$$x \mapsto P_{D_i}(x \lessgtr b_j) \text{ for } i \in \{0 \ldots 2n\}, \; x \in \{-\eta_1, \ldots, \eta_1\}$$

to the i-th variable node. As described in Sect. 4, for efficiency reasons, the computations of Eq. (3) are carried out in the Fourier domain.

The variable nodes combine incoming messages by computing the product with one index left out. For incoming messages $m_0, \ldots, m_l$ (m_j is the message coming from the j-th check node/inequality), the (normalized) product

$$x \mapsto \frac{\prod_{i \neq j} m_i(x)}{\sum_y \prod_{i \neq j} m_i(y)}, \; x \in \{-\eta_1, \ldots, \eta_1\}$$

is sent to the j-th check node.

Sending messages from variable nodes to factor nodes or vice-versa is called an *iteration*; sending messages from variable nodes to factor nodes and vice-versa is called a *full iteration*. After each full iteration, we may compute the current resulting probability distributions for the i-th coefficient, by multiplying and normalizing the incoming messages at the i-th node. That is, we compute the distribution

$$x \mapsto \frac{\prod_j m_j(x)}{\sum_y \prod_j m_j(y)},\ x \in \{-\eta_1, \ldots, \eta_1\}$$

and use this as (preliminary) result for the i-th coefficient. We do not use an equivalent of the clustering method used in [PP21].

Note that this process is also related to decoding Low-density parity-check codes using belief propagation [Mac03]. In our setting, the implied code is not low-density, instead, all variable nodes are connected to all factor nodes.

The belief propagation varies with the parameters k and η_1, depending on the parameter set as described in Table 1, and l. The parameter k determines the number of variable nodes, given by $2nk$; the size of a message is $2\eta_1 + 1$ probabilities. Each of the l observations corresponds to a check node.

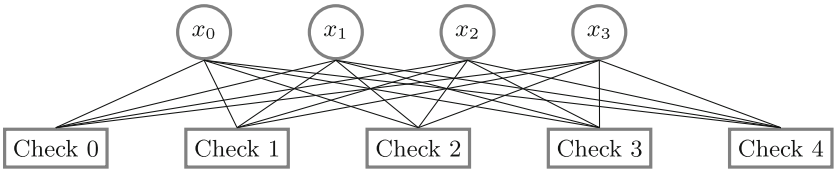

Fig. 2. A belief propagation graph with four variable nodes and five check nodes corresponding to four unknown coefficients and five inequalities.

4 Attack Implementation and Simulation

We implemented the simulation of the attack in Python and Rust where we rely on a modified version of PQClean [PQC]. We obtain inequalities by calling the Kyber implementation from Python, process them, and use a Rust implementation of belief propagation to recover the secret key from those inequalities. As single-bit faults are an established fault model and can be achieved e.g. using a laser, we solely rely on a simulation. The implementation is available at https://github.com/juliusjh/fault_enabled_cca.

4.1 Introducing Faults

To simulate our attack, we first fix a secret key on the assumed device. The simulated attacker then generates ciphertexts, checking for each ciphertext if the manipulation described in Sect. 3.1 only affects a single bit in the compressed

ciphertext. If this is the case, we call a manipulated decapsulation function, correcting the manipulated ciphertext before the comparison of the ciphertext with the re-encrypted ciphertext. This function corresponds to a call to the device under attack during which we flip one bit of the ciphertext. By observing whether the correct shared secret is returned, we retrieve an inequality as described in Sect. 3.2. These steps are repeated until a sufficient number of inequalities have been extracted. Note that ciphertexts that differ in more than one bit are not sent to the device but are discarded before any communication happens.

For our simulation, we assume a perfect bit-flipping fault. As mentioned in Sect. 3.1, this assumption can be relaxed at the expense of having to send more manipulated ciphertexts to the device under attack. Depending on the fault model, it might be favorable to allow the manipulated ciphertext to differ in multiple bits, e.g. in a pattern matching some property of the expected introduced fault.

4.2 Belief Propagation

For each coefficient of $\mathbf{e}$ and $\mathbf{s}$, we initialize a variable node and for each inequality, we initialize a check node. We then propagate for a maximum of 80 full iterations, i.e. from variable to factor nodes and vice-versa, and, after each full iteration, we retrieve the resulting probability distribution at each variable node and sort by

- entropy,
- min-entropy[6],
- entropy change since the last iteration,
- entropy and min-entropy.

If the first n coefficients are correct in any ordering, we can find the other n coefficients by solving the public key equation $\mathbf{b} = \mathbf{A}\mathbf{s}+\mathbf{e}$ using linear algebra. In this case, the belief propagation is aborted and counted as a success. To minimize the runtime for obtaining statistics, we also abort if after 5 full iterations, the number of correct coefficients has not improved.

To compute the distributions D_i, described in Sect. 3.3, Eq. (3), for each variable node i, the convolution of $n-1$ messages has to be computed. Naïvely, this results in having to compute $(n-1)n$ convolutions of distributions (with growing support) at each check node in every full iteration. To avoid inefficient re-computations when computing partial products in check nodes, we are using the binary tree algorithm described by Pessl and Prokop in [PP21]. Using an upwardly constructed and a downwardly constructed tree, we are avoiding recomputations and may compute all D_i at once. This is similar to the classical two-directional pass to compute partial products.

To construct the upward tree, we first initialize leaf nodes with $\hat{y}_i$. Every following layer then consists of the product of two nodes of the previous layer up until the $\log(2n)-1$-th layer having two nodes. The downward tree is then

[6] The negative logarithm of the probability of the most likely value.

initialized by swapping the values of the last layer of the upward tree. Every other layer of the downward tree is computed by multiplying the node of the layer above with the sibling from the upward tree. Thus, each node of the downward tree is the product of all nodes except its child nodes. The algorithm is also used for variable nodes. As the number of inequalities is not always a power of two, we add an additional node to a layer if the number nodes in the previous layer is not even. This additional node is equal to the last node of the previous layer, i.e. we implicitly add a node with the value of the multiplicative neutral element to each uneven layer.

Algorithm 7. Computations at a check node representing a less-equal inequality given by $a_0, \dots a_{2n-1}$, a value b, and set of possible values V. Messages m map a 16-bit signed value to a probability represented as 64-bit float by the $[\cdot]$ operator. The i-th messages represents the variable x_i corresponding to the coefficient a_i in the inequality.

Input: Incoming messages $m_0, \dots, m_{2n-1}$
Output: Outgoing messages $m'_0, \dots, m'_{2n-1}$

for all $i \in \{0, \dots, 2n-1\}$ **do**
  for all $v \in V$ **do**
    $mm_i[a_i \cdot v] \leftarrow m_i[x]$ ▷ Distribution of $a_i x_i$
  $\widehat{mm}_i \leftarrow \mathsf{FFT}(mm_i)$
$\mathsf{downtree} \leftarrow \mathsf{BinaryTrees.compute}(\widehat{mm}_0, \dots, \widehat{mm}_{2n-1})$ ▷ Multiply leaving one out
for all $i \in \{0, \dots, 2n-1\}$ **do**
  $\widehat{mp}_i \leftarrow \mathsf{downtree.leaf}(i)$ ▷ $\mathsf{downtree.leaf}(i) = \prod_{j \neq i} \widehat{mm}_j$
  $mp \leftarrow \mathsf{FFT}^{-1}(\widehat{mp}_i)$ ▷ Holds distribution of $\sum_{j \neq i} a_j x_j$
  for all $v \in V$ **do**
    $m'_i[v] \leftarrow mp_i.\mathsf{sum\_lesseq\_than}(\mathsf{b} - \mathsf{v})$ ▷ $m'_i[v] = P(\sum_{j \neq i} a_j x_j \leq b - v)$
return $m'_0, \dots, m'_{2n-1}$

4.3 Results

We ran our simulations for all three Kyber parameter sets to determine the number of inequalities and thus faulted decapsulations necessary to retrieve the secret key. We first determined the range of inequalities from which on a key recovery is possible. We than tested different numbers of inequalities with a step-size of 250, 500, and 1000 inequalities depending on the rate of change in the recovered coefficients in that range. For every number of inequalities, we ran 20 experiments, the success rate is then calculated as the number of successful runs divided by the total number of runs. The number of recovered coefficients is the average number of recovered coefficients.

Abort Criteria. We abort if the key is found or after at most 80 full iterations. To check for a successful recovery, an attacker would use the ordering described

in Sect. 4.2 and the public key equation. To obtain statistics, we simply abort if at least n of the coefficients in any of the orderings are correct. In addition, we abort early if there is no improvement after 20 steps.

Success Rate. Figure 3 shows the success rate of our experiments. In our simulations, our approach recovered the secret key in all cases starting with 5750 inequalities for Kyber512, 6750 inequalities for Kyber768, and 8500 inequalities for Kyber1024. Each inequality corresponds to one manipulated key exchange (and therefore one fault) with the device. More inequalities gave a higher success rate in all experiments. Succeeding runs usually occur starting with 5000 inequalities for Kyber512, 5750 inequalities for Kyber768, and 7250 inequalities for Kyber1024.

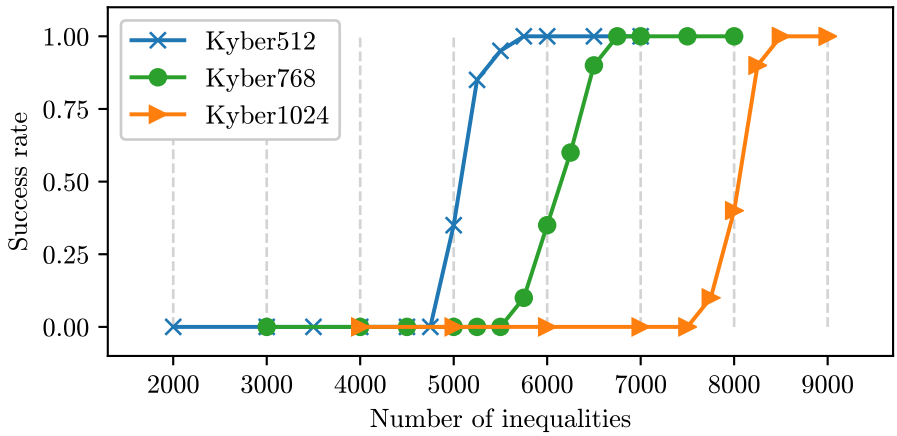

Fig. 3. Success rate depending on the number of inequalities/introduced faults.

Recovered Coefficients. In failing cases, we often recover a high number of coefficients that might be sufficient to retrieve the key, using standard lattice reduction algorithms such as [CN11]. We count recovered coefficients as the longest chain of correct coefficients in one of the orderings described in Sect. 4.2 after any step. As the number of correct coefficients might decrease after further iterations, especially in corner cases, an attacker should test if the key can be recovered using the public key equations after every step. If additional lattice reduction techniques are used, those should also be applied to intermediate results. Figure 4 shows the average number of recovered coefficients. Given that the number of recovered inequalities rises quickly after obtaining more than a certain threshold of inequalities, we assume that additional lattice reduction techniques are only useful for cases where obtaining further inequalities is very difficult or expensive. In all other cases where lattice reduction techniques may yield a correct result, retrieving more inequalities should quickly increase the success rate to 1.

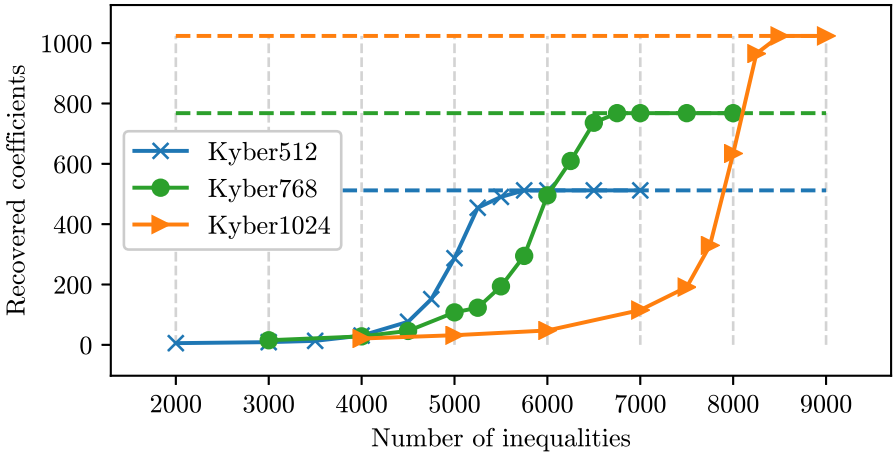

Fig. 4. Average number of recovered coefficients. The dashed lines mark 512, 768, and 1024 coefficients.

Runtime. Our implementation is in large parts multithreaded with the ability to utilize as many cores as the number of inequalities used. The runtime strongly depends on the number of inequalities as each inequality is represented by a factor node.

Our attack runs on widely available hardware. For example, on a standard laptop, one full iteration for Kyber768 and with 7000 inequalities takes about 15 min using a single thread of an Intel(R) Core(TM) i7-10510U CPU. Table 2 shows the average number of full iterations and the runtime in minutes for successful runs with 32 and 8 threads on a Intel(R) Xeon(R) Gold 6242 CPU with 16 cores.

Table 2. Runtimes in minutes on a Intel(R) Xeon(R) Gold 6242 with 32 and 8 threads.

Parameter set	Iterations	32 threads	8 threads
Kyber512 (6000 inequalities)	6.8	3.25	9.3
Kyber768 (7000 inequalities)	6.75	6.7	18.6
Kyber1024 (9000 inequalities)	9	16.9	39.25

Comparison with Previous Work. The work of Pessl and Prokop, presented in [PP21], uses a different attack (especially regarding the fault model) to obtain similar inequalities. Their technique to recover the secret key from those inequalities is different, but in a way related to ours.

Note that they analyzed an older version of Kyber. The main difference between the versions, in the context of key recovery from inequalities, is the

use of $\eta = 3$ (instead of $\eta = 2$ as in the previous version) for Kyber512. This probably makes recovering the secret key slightly harder in our case.

Pessl and Prokop report a success rate of 1 starting with 7500 inequalities for Kyber512, 10500 inequalities for Kyber768, and 11000 inequalities for Kyber1024. While lacking an exact comparison, our technique seems to be a clear improvement. We assume that our belief propagation based approach better avoids feedback loops which might negatively impact the result. Our implementation also uses significantly less memory than the implementation of [PP21]. While the original recovery technique requires up to 79 GB of RAM, we stay well below 10 GB of RAM usage at all times, depending on the number inequalities and threads, where we used up to 10000 inequalities and 40 threads. Regarding runtime, our implementation is slightly slower, but the attack may be carried out using a normal laptop.

5 Conclusion and Countermeasures

In the previous sections, we presented a realistic attack by combining a chosen-ciphertext attack with a fault injection and introduced an improved recovery technique for linear inequalities involving the secret key. The fault may be injected over a long execution-time interval and on different variables which hold public data. Our attack depends on the result of computations and not the computational steps itself. Therefore, securing computations and Boolean masking of inputs does not prevent our attack. We thereby highlighted the importance to protect seemingly non-sensitive, public data as well as operations over the whole execution time and to implement additional countermeasures to protect against fault-enabled chosen-ciphertext attacks. In addition, we give another exemplary usage of the belief propagation algorithm for key recovery and provide further evidence for the importance of the belief propagation algorithm for side-channel analysis. While we targeted Kyber, we conjecture that variants of this attacks are applicable to conceptually similar schemes such as Saber [DKRV18], FrodoKEM [ABD+21], or NewHope [AAB+19]. In this section, we provide an overview over standard countermeasures and if and how they defend against our attack.

Shuffling. In contrast to [PP21], our attack may not be mitigated by shuffling the decoder. They introduce a fault in the decoder and need to know which bit has been faulted to extract correct inequalities, which explains the usefulness of shuffling. Shuffling in time, however, does not affect the physical location of c in RAM, which is why manipulation of this value is not prevented. Still, a second attack path, namely manipulating c' during re-encryption (cf. Fig. 1), becomes more difficult to exploit. When only using successful decapsulations–they can only occur when the correct coefficient was faulted–the number of required faults is approximately multiplied by the number of shuffling positions.

Redundancy. Introducing redundancy in the storage of c, as depicted in Fig. 5, might drastically increase the effort and abilities required by an attacker. For that, a hash of c is computed directly after receiving the ciphertext. Right after the comparison check of the re-encryption, c is again hashed and compared against the previously computed hash. This mainly protects against faults introduced in c while it is stored in RAM. A manipulation during compression of c' is still possible, therefore the attack is not fully prevented but significantly harder to carry out.

The combination of introducing redundancy, shuffling the decoder, together with a secured comparison (via, e.g., double computation) likely prevents our attack with high probability. Additional security can be achieved by randomising the memory layout, e.g. by storing coefficients in a permuted order, and shuffling the compression function.

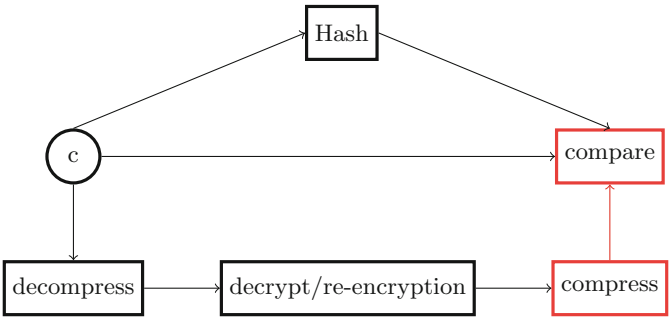

Fig. 5. Visualisation of the FO-transforms re-encryption check including decompression and compression with an additional countermeasure. Being able to introduce one-bit faults anywhere in the red (light) phases enables our attack. Introducing redundancy drastically reduces the attack surface. (Color figure online)

Acknowledgments. This work has been supported by the German Federal Ministry of Education and Research (BMBF) under the project "PQC4MED" (16KIS1041), as well as by the European Union's Horizon 2020 research and innovation program under grant agreement No 830927. We would like to thank the anonymous reviewers for their helpful comments which improved this work.

References

[AAB+19] Alkim, E., et al.: NewHope - Submission to the NIST post-quantum project (2019). https://newhopecrypto.org/data/NewHope_2019_07_10.pdf

[ABD+21] Alkim, E., et al.: FrodoKEM Learning With Errors Key Encapsulation (2021). https://frodokem.org/files/FrodoKEM-specification-20210604.pdf

[ACLZ20] Amiet, D., Curiger, A., Leuenberger, L., Zbinden, P.: Defeating NewHope with a single trace. In: Ding, J., Tillich, J.-P. (eds.) PQCrypto 2020. LNCS, vol. 12100, pp. 189–205. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-44223-1_11

[BDH+21] Bhasin, S., D'Anvers, J.-P., Heinz, D., Pöppelmann, T., Van Beirendonck, M.: Attacking and defending masked polynomial comparison for lattice-based cryptography. IACR Cryptology ePrint Archive **2021**, 104 (2021)

[BDK+18] Bos, J.W., et al.: CRYSTALS - Kyber: a CCA-secure module-lattice-based KEM. In: 2018 IEEE European Symposium on Security and Privacy, EuroS&P 2018, London, UK, 24–26 April 2018, pp. 353–367. IEEE (2018)

[BGRR19] Bauer, A., Gilbert, H., Renault, G., Rossi, M.: Assessment of the key-reuse resilience of NewHope. In: Matsui, M. (ed.) CT-RSA 2019. LNCS, vol. 11405, pp. 272–292. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-12612-4_14

[BPO+20] Bache, F., Paglialonga, C., Oder, T., Schneider, T., Güneysu, T.: High-speed masking for polynomial comparison in lattice-based KEMs. TCHES **2020**(3), 483–507 (2020)

[CN11] Chen, Y., Nguyen, P.Q.: BKZ 2.0: better lattice security estimates. In: Lee, D.H., Wang, X. (eds.) ASIACRYPT 2011. LNCS, vol. 7073, pp. 1–20. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-25385-0_1

[DKRV18] D'Anvers, J.-P., Karmakar, A., Sinha Roy, S., Vercauteren, F.: Saber: module-LWR based key exchange, CPA-secure encryption and CCA-secure KEM. In: Joux, A., Nitaj, A., Rachidi, T. (eds.) AFRICACRYPT 2018. LNCS, vol. 10831, pp. 282–305. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-89339-6_16

[FO99] Fujisaki, E., Okamoto, T.: Secure integration of asymmetric and symmetric encryption schemes. In: Wiener, M. (ed.) CRYPTO 1999. LNCS, vol. 1666, pp. 537–554. Springer, Heidelberg (1999). https://doi.org/10.1007/3-540-48405-1_34

[GGSB20] Guo, Q., Grosso, V., Standaert, F.-X., Bronchain, O.: Modeling soft analytical side-channel attacks from a coding theory viewpoint. IACR TCHES **2020**(4), 209–238 (2020)

[GJN20] Guo, Q., Johansson, T., Nilsson, A.: A key-recovery timing attack on post-quantum primitives using the Fujisaki-Okamoto transformation and its application on FrodoKEM. IACR Cryptology ePrint Archive **2020**, 743 (2020)

[GRO18] Green, J., Roy, A., Oswald, E.: A systematic study of the impact of graphical models on inference-based attacks on AES. In: Bilgin, B., Fischer, J.-B. (eds.) CARDIS 2018. LNCS, vol. 11389, pp. 18–34. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-15462-2_2

[HHK17] Hofheinz, D., Hövelmanns, K., Kiltz, E.: A modular analysis of the Fujisaki-Okamoto transformation. In: Kalai, Y., Reyzin, L. (eds.) TCC 2017. LNCS, vol. 10677, pp. 341–371. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-70500-2_12

[HHP+21] Hamburg, M., et al.: Chosen ciphertext k-trace attacks on masked CCA2 secure kyber. IACR Cryptology ePrint Archive **2021**, 956 (2021)

[HP21] Heinz, D., Pöppelmann, T.: Combined fault and DPA protection for lattice-based cryptography. IACR Cryptology ePrint Archive **2021**, 101 (2021)

[KPP20] Kannwischer, M.J., Pessl, P., Primas, R.: Single-trace attacks on Keccak. TCHES **2020**(3), 243–268 (2020)

[LPR13] Lyubashevsky, V., Peikert, C., Regev, O.: On ideal lattices and learning with errors over rings. J. ACM **60**(6), 43:1-43:35 (2013)

[LS15] Langlois, A., Stehlé, D.: Worst-case to average-case reductions for module lattices. Des. Codes Crypt. **75**(3), 565–599 (2014). https://doi.org/10.1007/s10623-014-9938-4

[Mac03] MacKay, D.J.C.: Information Theory, Inference, and Learning Algorithms. Cambridge University Press, Cambridge (2003)

[Nata] National Institute of Standards and Technology. NIST Status Update on the 3rd Round. https://csrc.nist.gov/CSRC/media/Presentations/status-update-on-the-3rd-round/images-media/session-1-moody-nist-round-3-update.pdf

[Natb] National Institute of Standards and Technology. Post-Quantum Cryptography Standardization. https://csrc.nist.gov/Projects/Post-Quantum-Cryptography/Post-Quantum-Cryptography-Standardization

[OGM17] Ordas, S., Guillaume-Sage, L., Maurine, P.: Electromagnetic fault injection: the curse of flip-flops. J. Cryptogr. Eng. **7**(3), 183–197 (2016). https://doi.org/10.1007/s13389-016-0128-3

[OSPG18] Oder, T., Schneider, T., Pöppelmann, T., Güneysu, T.: Practical CCA2-secure and masked Ring-LWE implementation. TCHES **2018**(1), 142–174 (2018)

[PH16] Park, A., Han, D.-G.: Chosen ciphertext Simple Power Analysis on software 8-bit implementation of Ring-LWE encryption. In: 2016 IEEE Asian Hardware-Oriented Security and Trust, AsianHOST 2016, Yilan, Taiwan, 19–20 December 2016, pp. 1–6. IEEE Computer Society (2016)

[PP19] Pessl, P., Primas, R.: More practical single-trace attacks on the number theoretic transform. In: Schwabe, P., Thériault, N. (eds.) LATINCRYPT 2019. LNCS, vol. 11774, pp. 130–149. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-30530-7_7

[PP21] Pessl, P., Prokop, L.: Fault attacks on CCA-secure lattice KEMs. TCHES **2021**(2), 37–60 (2021)

[PPM17] Primas, R., Pessl, P., Mangard, S.: Single-trace side-channel attacks on masked lattice-based encryption. In: Fischer, W., Homma, N. (eds.) CHES 2017. LNCS, vol. 10529, pp. 513–533. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-66787-4_25

[PQC] Contributors to PQClean. PQClean. https://github.com/PQClean/PQClean

[RBRC20] Ravi, P., Bhasin, S., Roy, S.S., Chattopadhyay, A.: Drop by Drop you break the rock - Exploiting generic vulnerabilities in Lattice-based PKE/KEMs using EM-based Physical Attacks. IACR Cryptology ePrint Archive, p. 549 (2020)

[RRCB20] Ravi, P., Roy, S.S., Chattopadhyay, A., Bhasin, S.: Generic side-channel attacks on CCA-secure lattice-based PKE and KEMs. TCHES **2020**(3), 307–335 (2020)

[RRdC+16] Reparaz, O., Roy, S.S., de Clercq, R., Vercauteren, F., Verbauwhede, I.: Masking Ring-LWE. J. Cryptogr. Eng. **6**(2), 139–153 (2016)

[RRVV15] Reparaz, O., Sinha Roy, S., Vercauteren, F., Verbauwhede, I.: A masked Ring-LWE implementation. In: Güneysu, T., Handschuh, H. (eds.) CHES 2015. LNCS, vol. 9293, pp. 683–702. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-48324-4_34

[RSDT13] Roscian, C., Sarafianos, A., Dutertre, J.-M., Tria, A.: Fault model analysis of laser-induced faults in SRAM memory cells. In: Fischer, W., Schmidt, J.-M. (eds.) 2013 Workshop on Fault Diagnosis and Tolerance in Cryptography, Los Alamitos, CA, USA, 20 August 2013, pp. 89–98. IEEE Computer Society (2013)

[VGS14] Veyrat-Charvillon, N., Gérard, B., Standaert, F.-X.: Soft analytical side-channel attacks. In: Sarkar, P., Iwata, T. (eds.) ASIACRYPT 2014. LNCS, vol. 8873, pp. 282–296. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-662-45611-8_15

[VOGR18] Valencia, F., Oder, T., Güneysu, T., Regazzoni, F.: Exploring the vulnerability of R-LWE encryption to fault attacks. In: Goodacre, J., Luján, M., Agosta, G., Barenghi, A., Koren, I., Pelosi, G. (eds.) Proceedings of the Fifth Workshop on Cryptography and Security in Computing Systems, CS2 2018, Manchester, UK, 24 January 2018, pp. 7–12. ACM (2018)

[XIU+21] Xagawa, K., Ito, A., Ueno, R., Takahashi, J., Homma, N.: Fault-injection attacks against NIST's post-quantum cryptography round 3 KEM candidates. IACR Cryptology ePrint Archive **2021**, 840 (2021)