Fault-Injection Attacks Against NIST's Post-Quantum Cryptography Round 3 KEM Candidates

Keita Xagawa[3(✉)], Akira Ito[1(✉)], Rei Ueno[1,2(✉)], Junko Takahashi[3(✉)], and Naofumi Homma[1,4(✉)]

[1] Tohoku University, 2–1–1 Katahira, Aoba-ku, Sendai-shi 980-8577, Japan
{ito,homma}@riec.tohoku.ac.jp, rei.ueno.a8@tohoku.ac.jp

[2] PRESTO, Japan Science and Technology Agency, 4–1–8 Honcho, Kawaguchi, Saitama 332-0012, Japan

[3] NTT Social Informatics Laboratories, 3-9-11 Midori-cho, Musashino-shi, Tokyo 180-8585, Japan
{keita.xagawa.zv,junko.takahashi.fc}@hco.ntt.co.jp

[4] CREST, Japan Science and Technology Agency, 4–1–8 Honcho, Kawaguchi, Saitama 332-0012, Japan

Abstract. We investigate *all* NIST PQC Round 3 KEM candidates from the viewpoint of fault-injection attacks: Classic McEliece, Kyber, NTRU, Saber, BIKE, FrodoKEM, HQC, NTRU Prime, and SIKE. All KEM schemes use variants of the Fujisaki-Okamoto transformation, so the equality test with re-encryption in decapsulation is critical.

We survey effective key-recovery attacks when we can skip the equality test. We found the existing key-recovery attacks against Kyber, NTRU, Saber, FrodoKEM, HQC, one of two KEM schemes in NTRU Prime, and SIKE. We propose a new key-recovery attack against the other KEM scheme in NTRU Prime. We also report an attack against BIKE that leads to leakage of information of secret keys.

The open-source pqm4 library contains all KEM schemes except Classic McEliece and HQC. We show that giving a single instruction-skipping fault in the decapsulation processes leads to skipping the equality test *virtually* for Kyber, NTRU, Saber, BIKE, and SIKE. We also report the experimental attacks against them. We also report the implementation of NTRU Prime allows chosen-ciphertext attacks freely and the timing side-channel of FrodoKEM reported in Guo, Johansson, and Nilsson (CRYPTO 2020) remains, while there are no such bugs in their NIST PQC Round 3 submissions.

Keywords: Post-quantum cryptography · NIST PQC standardization · KEM · The Fujisaki-Okamoto transformation · Fault-injection attacks

M. Tibouchi and H. Wang (Eds.): ASIACRYPT 2021, LNCS 13091, pp. 33–61, 2021.
https://doi.org/10.1007/978-3-030-92075-3_2

1 Introduction

1.1 Background

Key encapsulation mechanism: Public-key encryption (PKE in short) allows us to send a message secretly without a pre-shared secret key [30,67,73], which is a fundamental task of cryptography. PKE consists of three algorithms; a key-generation algorithm that generates a public key and a secret key, an encryption algorithm that takes a message and a public key as input and outputs a ciphertext, and a decryption algorithm that takes a secret key and a ciphertext as input and outputs a message.

Key encapsulation mechanism (KEM in short) is also fundamental cryptographic primitive [1,26,72], which can be considered as a variant of public-key encryption (PKE). KEM's encryption algorithm, which we call the encapsulation algorithm, takes a public key as input and outputs a ciphertext and *a key* (or an ephemeral key). KEM's decryption algorithm, which we call the decapsulation algorithm, takes a secret key and a ciphertext as input and outputs a key instead of a message. KEM's sender and receiver share a key instead of a message in the case of PKE. KEM is a versatile primitive and has a lot of applications, e.g., key exchange, hybrid encryption, secure authentication, and authenticated key exchange.

The most standard security notion of KEM is indistinguishability against chosen-ciphertext attacks (IND-CCA security) [26,63]. Since it is hard to construct efficient IND-CCA-secure KEMs directly, cryptographers often use the transformations from weakly-secure PKE/KEM into IND-CCA-secure KEM. The Fujisaki-Okamoto (FO) transformation [29,35,36] is one of the transformations often used in the design of IND-CCA-secure PKE/KEM in the random oracle model (ROM). Roughly speaking, the FO transformation transforms an underlying PKE scheme into KEM as follows: Let G and H be two random oracles. A key-generation algorithm of KEM is that of PKE. An encapsulation algorithm on input a public key pk chooses a message m randomly, encrypts it into $ct = \mathsf{Enc}(pk, m; \mathsf{G}(m))$, where Enc is an encryption algorithm of PKE and the randomness of encryption is computed as $\mathsf{G}(m)$, and outputs a ciphertext ct and a key $K = \mathsf{H}(m)$. A decapsulation algorithm on input sk and ct decrypts ct into $m' = \mathsf{Dec}(sk, ct)$, where Dec is a decryption algorithm of PKE, re-encrypts m' into $ct' = \mathsf{Enc}(pk, m'; \mathsf{G}(m'))$, and outputs a key $K = \mathsf{H}(m')$ if $ct = ct'$ and a rejection symbol otherwise.

Post-quantum cryptography: Scalable quantum computers will threaten classical public-key cryptography since Shor's algorithm on a quantum machine solves factorization and discrete logarithms efficiently [71]. The recent progress in developing quantum machines motivates us to replace classical public-key cryptography with post-quantum cryptography (PQC). Hence, in the past decade, the security proofs of the FO transformation and its variants have been extended to those in the quantum random oracle model (QROM) [19] to show the security against *quantum* polynomial-time adversary. See e.g., [17,42,47,48,52,69,76].

Moreover, in 2016, NIST PQC standardization called for proposals on PKE/KEM and signatures as the basic primitives[1]. In 2020, NIST selected four finalists and five alternate candidates for KEM in Round 3 [6]. All use the FO-like transformations to construct IND-CCA-secure KEMs in the (Q)ROM.

Fault-injection attacks: In the real world, the decapsulation algorithm is implemented physically. Hence, investigations into side-channel attacks (SCA) [50,51] and fault-injection attacks (FIA) [16,20] against proposed KEMs are strongly promoted by NIST. The attacks' targets are recovery of an ephemeral key of a given ciphertext or a secret key of a given public key. We call the former and the latter ephemeral-key-recovery attack and key-recovery attack, respectively.

We focus on FIA against KEM and review the scenario of it. Suppose that an adversary can inject faults into a decapsulation machine that contains a secret key. In this situation, it is natural to think the adversary has the machine itself (e.g., decapsulation machines in card, sensor, robot, and TV box) and uses it freely because the adversary can physically access the machine. Hence, the adversary can decrypt any ciphertexts and recover the corresponding ephemeral key of the target ciphertext. Thus, if we consider FIA, ephemeral-key-recovery attacks are not so important.

On the other hand, recovery of secret key via FIA is non-trivial and interesting, because the key-recovery attack logically breaks a tamper-resilient memory by extracting the secret from it. In addition, once one obtains a secret key from a decapsulation machine, one can copy the machine. Thus, we examine how FIA leads to a key-recovery attack.

There are a lot of techniques to make decapsulation faulty; shooting a LASER to set/reset a bit of SRAM [74], injecting a clock or power glitch [11,33,68], using electromagnetic (EM) pulses [41]. (Un)fortunately, an injection of fault often fails to obtain an expected result, say, a skip of an instruction of the assembly code. Thus, the less number of faults in a single run of decapsulation an attack requires, the better. Especially, we are interested in *single-fault* key-recovery attacks.

Skipping-the-equality-test attack: In the FO-like transformations, the decapsulation algorithm given a ciphertext ct first decrypts the ciphertext into m', re-encrypts it into ct', and returns $K = \mathsf{H}(m', \mathsf{aux})$ if $ct = ct'$ and pseudorandom value $K = \mathsf{H}(s, \mathsf{aux})$ or the rejection symbol $\perp$ otherwise, where H is a hash function modeled by a random oracle, aux depends on pk and ct, and s is a secret value.

By injecting a fault carefully, we could force the decapsulation machine to *skip* the equality test $ct = ct'$ and return $K = \mathsf{H}(m', \mathsf{aux})$ always, where $m' = \mathsf{Dec}(sk, ct)$. This enables us to implement a plaintext-checking oracle on input guess m_{guess} and ciphertext ct by checking if $K = \mathsf{H}(m_{\mathsf{guess}}, \mathsf{aux})$ or not and a key-mismatch oracle on input guess K_{guess} and ciphertext ct by checking if

[1] https://csrc.nist.gov/Projects/post-quantum-cryptography/post-quantum-cryptography-standardization/Call-for-Proposals.

$K = K_{\mathsf{guess}}$ or not. Such oracles would enable an adversary to mount a key-recovery attack against KEM.

Fault-injection attack against pre-quantum KEMs: Factoring/RSA-based PKE/KEM is vulnerable against FIA. For example, safe-error attacks [79,80] are effective to guess a bit of secret key. They are also applicable to Discrete-logarithm (DL)-based PKE/KEM. DL-based PKE/KEM has several attack surfaces vulnerable to FIA. See, for example, invalid point/curve attacks [8,15,18,75].

We note that the existing key-recovery FIAs do not target the equality test of the FO transformation. It is not known whether this plaintext-checking/key-mismatch oracle (or even decryption oracle) enables us to recover the secret key of the underlying PKE, say, the textbook RSA. (See e.g., [21] and [3].) Thus, the key-recovery FIA against pre-quantum KEMs that skips the equality test are not so explored.

Fault-injection attack against post-quantum KEMs: This situation is changed in post-quantum KEMs. Unfortunately, underlying PKEs in the post-quantum PKE/KEMs are often vulnerable to key-recovery chosen-ciphertext attacks. For example, Hall, Goldberg, and Schneier [40] pointed out message-recovery and key-recovery chosen-ciphertext attacks against the McEliece PKE [55,58] and the Ajtai-Dwork PKE [5], respectively. Fluhrer pointed out that a simple key-exchange scheme based on ring learning with errors (RLWE) is vulnerable to the key-mismatch attack if a user fixes its secret [34]. Galbraith, Petit, Shani, and Ti [37] gave a key-recovery key-mismatch attack against SIDH [28,46] with fixed secret. Therefore, the equality test is an important target of FIA.

Although Pessl and Prokop [60] pointed out that the equality test is '*an obvious faulting target*,' we do not know how easily we can mount a skipping-the-equality-test attack by injecting *a single fault* against the implementations in the wild and how effective the skipping-the-equality-test attack is against the NIST PQC Round 3 KEM candidates.

1.2 Our Contribution

We systematically study how effective fault-injection attacks that lead to the skip of the equality test of FO-like transformations are against *all* KEMs in the NIST PQC Round 3 finalists and the alternates: Classic McEliece [7], Kyber [70], NTRU (ntruhps and ntruhrss) [22], Saber [27], BIKE [9], FrodoKEM [57], HQC [4], NTRU Prime (sntrupr and ntrulpr) [14], and SIKE [45]. We summarize our findings in Table 1.

Theoretical analysis: We study whether the underlying PKEs of KEMs are resilient to key-recovery plaintext-checking attacks (KR-PCA) or not, since skipping the equality test enables an adversary to obtain $K = \mathsf{H}(\mathsf{Dec}(sk, ct), \mathsf{aux})$ instead of pseudorandom string or $\perp$ and to implement a plaintext-checking oracle easily.

Table 1. Summary of our findings on NIST PQC Round 3 KEM Candidates (finalists and alternates) and their implementations in pqm4: PCA implies plaintext-checking attack.

Name	Effect of PCA	Attack Surface in pqm4	Effect of FIA in pqm4
Classic McEliece [7]	Unknown	N/A	N/A
Kyber [70]	Key recovery	Skip	Key recovery
NTRU – ntruhps [22]	Key recovery	Skip	Key recovery
NTRU – ntruhrss [22]	Key recovery	Skip	Key recovery
Saber [27]	Key recovery	Skip	Key recovery
BIKE [9]	Key leakage (New)	Skip	Key leakage
FrodoKEM [57]	Key recovery	Timing bug	Key recovery
HQC [4]	Key recovery	N/A	N/A
NTRU Prime – sntrupr [14]	Key recovery	CCA bug	Key recovery
NTRU Prime – ntrulpr [14]	Key recovery (New)	CCA bug	Key recovery
SIKE [45]	Key recovery	Skip	Key recovery

We found that almost all PKEs except the underlying PKE of Classic McEliece leaks information of the decryption key in the presence of plaintext-checking oracle *in vitro*. Our findings are summarized as follows (see also Table 2):

Kyber, NTRU, Saber, FrodoKEM, HQC, sntrupr of NTRU Prime, and SIKE: We survey the literature and found that there are KR-PCAs against the underlying PKEs of Kyber, ntruhps and ntruhrss of NTRU, Saber, FrodoKEM, HQC, sntrupr of NTRU Prime, and SIKE.

ntrulpr of NTRU Prime: We propose a KR-PCA against the underlying PKE of NTRU LPRime (ntrulpr of NTRU Prime) by mimicking the KR-PCAs against the underlying PKEs of Saber and Kyber [44]. See Sect. 4.

BIKE: The underlying PKE of BIKE in round 3 also leaks the secret key's information from the plaintext-checking oracle as QC-MDPC [56] is vulnerable to the KR-PCA proposed by Guo, Johansson, and Stankvoski [39]. However, the change of a decoder algorithm in round 3 makes key-recovery attacks difficult. See the full version.

Classic McEliece: There are no known KR-PCAs against the underlying PKE of Classic McEliece if the decoder in a decryption algorithm rejects invalid plaintexts[2] (We note that the specifications seem to allow the use of any decoder that decodes binary Goppa codes.)

Trade-off: Skipping the equality test enables the adversary to obtain $K = \mathsf{KDF}(m, \mathsf{aux})$ with $m = \mathsf{Dec}(sk, ct)$ rather than the plaintext-checking oracle. Thus, the adversary can check if $m = m_{\mathsf{guess}}$ by checking $K = \mathsf{KDF}(m_{\mathsf{guess}}, \mathsf{aux})$ from *a single faulty experiment*. If the number of candidates of m is small, then

[2] The plaintext space is a set of n-dimensional vectors whose Hamming weight is t.

Table 2. Theoretical plaintext-checking attacks and key-mismatch attacks against the underlying PKEs of NIST PQC Round 3 KEM Candidates.

Name	Results
Classic McEliece [7]	Unknown
Kyber [70]	Key recovery [44,61,62,66]
NTRU – ntruhps [22]	Key recovery [31]
NTRU – ntruhrss [22]	Key recovery [81]
Saber [27]	Key recovery [44,62]
BIKE [9]	Key leakage (New, adapted from [39])
FrodoKEM [57]	Key recovery [10,62,66,77]
HQC [4]	Key recovery [44]
NTRU Prime – sntrupr [14]	Key recovery [64]
NTRU Prime – ntrulpr [14]	Key recovery (New, adapted from [10,44,61,62,66,77])
SIKE [45]	Key recovery [37]

we can determine the value of m by an exhaustive search. By using this property, there are trade-offs between the computational cost and the number of faulty experiments in the cases of Kyber, Saber, FrodoKEM, and ntrulpr of NTRU Prime. See the details in Sect. 4 for the case of ntrulpr of NTRU Prime.

Investigation of KEMs in pqm4: We investigate implementation of KEMs in pqm4 [49], which include Kyber, NTRU (ntruhps and ntruhrss), Saber, BIKE, FrodoKEM, NTRUPrime (sntrupr and ntrulpr), and SIKE[3]

- **NTRU Prime:** In the pqm4 implementation of NTRU Prime (sntrupr and ntrulpr), a decapsulation program contains a fatal bug that forces the result of the equality test to be true.[4] Thus, we can mount a chosen-ciphertext attack against them freely. See Subsect. 5.1.
- **FrodoKEM:** In 2020, Guo et al. [38] pointed out that the implementation of FrodoKEM (and HQC) contains a leaky equality test that leaks information of the secret key from the timing side channel and succeeded in mounting a key-recovery attack using the timing information. Although FrodoKEM in Round 3 repaired this leaky equality test, the bug still remains in the pqm4 implementation.[5] See Subsect. 5.2.
- **Kyber, NTRU, and Saber:** They shared a same structure to compute a key. Roughly speaking, decapsulation programs use a flag for the equality test and overwrite the decrypted result m' by a secret seed s if the flag is set. This overwriting is done by a single function call of ‘cmov’ (conditional-move). (Un)fortunately, we can skip this function call by a single fault and mount FIA. See Subsect. 5.3

[3] We use 2021 Jun. 5 version. https://github.com/mupq/pqm4/commit/8d3384d879b10619c8c36947e4be6ab13ec6d268.

[4] We report it in https://github.com/mupq/pqm4/issues/195.

[5] pqm4 noticed this issue. See https://github.com/mupq/pqm4/issues/161.

BIKE: The decapsulation program of BIKE in pqm4 computes mask, which is -1 or 0 depending on the re-encryption check, and overwrites the decryption result m' by a secret seed s or keep it as "$m' \leftarrow (m' \wedge \neg\mathsf{mask}) \vee (s \wedge \mathsf{mask})$".[6] (Un)fortunately, we identify a single operation such that if we skip the operation, then mask is set to 0 always. Thus, we can skip the overwrite procedure virtually by a single fault. See Subsect. 5.4.

SIKE: The implementation of SIKE in pqm4 simply uses an 'if' statement to overwrite the decrypted result m' by a secret seed s. In the assembly code, this if-then-overwrite is implemented as 'compare' and 'conditional jump'. (Un)fortunately, we can skip this 'conditional jump' by a single fault. See Subsect. 5.5.

Experimental results: On the basis of our findings, we conduct the experimental skip attacks on Kyber, NTRU, Saber, BIKE, and SIKE. The target is STM32F415 whose core is ARM Cortex-M4, which is a de-facto standard platform as NIST suggested. We run 100 fault injections to each scheme and succeeded in injecting faults correctly with 15–50%. See Sect. 6.

1.3 Related Works

For PQC candidates and their implementation, we recommend the reader to read an exhaustive survey written by Howe, Prest, and Apon [43]. Ravi and Roy gave a lecture on SCAs and FIAs against lattice-based PQC candidates [65]. Costello wrote a survey of isogeny-based cryptography [25]. For SCA, FIA, and key-recovery plaintext-checking/key-mismatch attacks against NIST PQC KEM Candidates, see our survey in the full version.

1.4 Organization

Section 2 reviews basic notions and notations. Section 3 reviews the variants of the FO transformation. Section 4 gives a key-recovery attack against ntrulpr of NTRU Prime using plaintext-checking oracle and discusses a trade-off between efficiency and the number of queries if we consider the fault-injection attack. Section 5 describes the equality test of KEMs and how we can mount skipping attack. Section 6 reports our experimental results. In the full version, we will review the variants of the FO transformation, the KEM schemes, and KR-PCAs against them. In addition, we will report our key-leakage PCAs against BIKE.

2 Preliminaries

2.1 Notation

A security parameter is denoted by λ. We use the standard O-notations. DPT, PPT, and QPT stand for deterministic polynomial-time, probabilistic polynomial-time, and quantum polynomial-time, respectively. A function $f(\lambda)$ is said to be *negligible* if $f(\lambda) = \lambda^{-\omega(1)}$. We denote a set of negligible functions

[6] If $\mathsf{mask} = 0$, then we have $m' \leftarrow m'$. Otherwise, we have $m' \leftarrow s$.

by $\mathsf{negl}(\lambda)$. For a statement P (e.g., $r \in [0,1]$), we define $\mathsf{boole}(P) = 1$ if P is satisfied and 0 otherwise.

For a distribution χ, we often write "$x \leftarrow \chi$," which indicates that we take a sample x in accordance with χ. For a finite set S, $U(S)$ denotes the uniform distribution over S. We often write "$x \leftarrow S$" instead of "$x \leftarrow U(S)$." If inp is a string, then "$\mathsf{out} \leftarrow \mathsf{A}(\mathsf{inp})$" denotes the output of algorithm A when run on input inp. If A is deterministic, then out is a fixed value and we write "$\mathsf{out} := \mathsf{A}(\mathsf{inp})$." We use the notation "$\mathsf{out} := \mathsf{A}(\mathsf{inp}; r)$" to make the randomness r explicit.

For an odd positive integer q, we define $r' := r \bmod^{\pm} q$ to be the unique element $r' \in [-(q-1)/2, (q-1)/2]$ with $r' \equiv r \pmod{q}$.

2.2 Public-Key Encryption (PKE)

The model for PKE schemes is summarized as follows:

Definition 2.1. *A PKE scheme* PKE *consists of the following triple of polynomial-time algorithms* $(\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$*:*

- $\mathsf{Gen}(1^\lambda; r_g) \to (pk, sk)$*: a key-generation algorithm that takes as input* 1^λ*, where* λ *is the security parameter, and randomness* $r_g \in \mathcal{R}_{\mathsf{Gen}}$ *and outputs a pair of keys* (pk, sk)*.* pk *and* sk *are called the encryption key and decryption key, respectively.*
- $\mathsf{Enc}(pk, m; r_e) \to ct$*: an encryption algorithm that takes as input encryption key* pk*, message* $m \in \mathcal{M}$*, and randomness* $r_e \in \mathcal{R}_{\mathsf{Enc}}$ *and outputs ciphertext* $ct \in \mathcal{C}$*.*
- $\mathsf{Dec}(sk, ct) \to m/\bot$*: a decryption algorithm that takes as input decryption key* sk *and ciphertext* ct *and outputs message* $m \in \mathcal{M}$ *or a rejection symbol* $\bot \notin \mathcal{M}$*.*

Definition 2.2. *We say a PKE scheme* PKE *is deterministic if* Enc *is deterministic, that is, it takes* pk *and* m *and does not take a randomness* r_e*. DPKE stands for deterministic public-key encryption.*

Plaintext-checking oracle: Since we review and propose key-recovery attacks using plaintext-checking oracle (PCO), we formally review the definition of the plaintext-checking oracle [2,59].

Definition 2.3 (Plaintext-Checking Oracle). *A plaintext-checking oracle* PCO *takes as input a plaintext* m *and a ciphertext* ct *and outputs* 1 *if and only if* m *is equal to the decrypted result* $\mathsf{Dec}(sk, ct)$*. That is,* $\mathrm{PCO}(m, ct) := \mathsf{boole}(m = \mathsf{Dec}(sk, ct))$*.*

2.3 Key Encapsulation Mechanism (KEM)

The model for KEM schemes is summarized as follows:

Definition 2.4. *A KEM scheme* KEM *consists of the following triple of polynomial-time algorithms* $(\mathsf{Gen}, \mathsf{Encaps}, \mathsf{Decaps})$*:*

– $\mathsf{Gen}(1^\lambda; r_g) \to (pk, sk)$*: a key-generation algorithm that takes as input* 1^λ*, where* λ *is the security parameter, and randomness* $r_g \in \mathcal{R}_{\mathsf{Gen}}$ *and outputs a pair of keys* (pk, sk)*.* pk *and* sk *are called the encapsulation key and decapsulation key, respectively.*
– $\mathsf{Encaps}(pk; r_e) \to (ct, K)$*: an encapsulation algorithm that takes as input encapsulation key* pk *and randomness* $r_e \in \mathcal{R}_{\mathsf{Encaps}}$ *and outputs ciphertext* $ct \in \mathcal{C}$ *and key* $K \in \mathcal{K}$*.*
– $\mathsf{Decaps}(sk, ct) \to K/\bot$*: a decapsulation algorithm that takes as input decapsulation key* sk *and ciphertext* ct *and outputs key* K *or a rejection symbol* $\bot \notin \mathcal{K}$*.*

Key-mismatch oracle: We review the key-mismatch oracle, which is an analogue of the plaintext-checking oracle for PKE.

Definition 2.5. [Key-Mismatch Oracle]. *A key-mismatch oracle* KMO *takes as input a key* K *and a ciphertext* ct *and outputs* 1 *if and only if* K *is equal to the decapsulated result* $\mathsf{Decaps}(sk, ct)$*. That is,* $\mathrm{KMO}(K, ct) := \mathsf{boole}(K = \mathsf{Decaps}(sk, ct))$*.*

3 Variants of the Fujisaki-Okamoto Transformation

We review the variants of the FO transformation that are used by NIST PQC Round 3 candidate KEMs: $\mathsf{FO}^{\not\perp}$ in this section and $\mathsf{FO}^{\not\perp\prime}$, $\mathsf{FO}^{\not\perp\prime\prime}$, $\mathsf{HFO}^{\perp}$, $\mathsf{HFO}^{\not\perp}$, SXY, and $\mathsf{HU}^{\not\perp}$ in the full version. Let $\mathsf{PKE} = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ be a PKE, whose ciphertext space is $\mathcal{C}_{\mathsf{PKE}}$. If PKE is probabilistic, then $\mathcal{R}_{\mathsf{Enc}}$ denotes the randomness space of Enc. Let $\{0,1\}^{k(\lambda)}$ be the key space of KEM.

3.1 FO with Implicit Rejection

$\mathsf{FO}^{\not\perp}$ transforms a weakly-secure probabilistic PKE into IND-CCA-secure KEM, where the identifier "$\not\perp$" implies *implicit rejection* [42]. This variant is used by BIKE and SIKE.

Let $\{0,1\}^{\ell(\lambda)}$ be the plaintext space of PKE. Let $\mathsf{G}\colon \{0,1\}^* \to \mathcal{R}_{\mathsf{Enc}}$ and $\mathsf{H}\colon \{0,1\}^{\ell(\lambda)} \times \mathcal{C}_{\mathsf{PKE}} \to \{0,1\}^{k(\lambda)}$ be hash functions modeled by the random oracles. The $\mathsf{FO}^{\not\perp}$ is summarized as Fig. 1. Assuming the IND-CPA security of PKE, the obtained KEM scheme is IND-CCA-secure in the QROM (see e.g., [52]).

Remark 3.1. BIKE and SIKE do *not* test whole re-encryption check. Roughly speaking, their encryption algorithm Enc is separable into two algorithms Enc_1 and Enc_2. Enc_1 takes pk and randomness r and outputs c_1 and $k \in \{0,1\}^{\ell(\lambda)}$. Enc_2 takes m and k and outputs $c_2 := k \oplus m$.

Using this property, BIKE omits the re-encryption check. Concretely speaking, k in BIKE's Enc_1 is computed as $k := \mathsf{H}(r)$, where H is a hash function modeled by the random oracle. BIKE's Dec internally obtains r' and checks the

$\mathsf{Gen}(1^\lambda)$	$\mathsf{Encaps}(pk)$	$\mathsf{Decaps}(\overline{sk}, ct)$, where $\overline{sk} = (sk, pk, s)$
$(pk, sk) \leftarrow \mathsf{Gen}(1^\lambda)$	$m \leftarrow \{0,1\}^{\ell(\lambda)}$	$m' := \mathsf{Dec}(sk, ct)$
$s \leftarrow \{0,1\}^{\ell(\lambda)}$	$r := \mathsf{G}(m)$ // for BIKE	$r' := \mathsf{G}(m')$ // for BIKE
$\overline{sk} := (sk, pk, s)$	$r := \mathsf{G}(m, pk)$ // for SIKE	$r' := \mathsf{G}(m', pk)$ // for SIKE
return $(pk, \overline{sk})$	$ct := \mathsf{Enc}(pk, m; r)$	$ct' := \mathsf{Enc}(pk, m'; r')$
	$K := \mathsf{H}(m, ct)$	**if** $ct = ct'$, **then return** $K := \mathsf{H}(m', ct)$
	return (K, ct)	**else return** $K := \mathsf{H}(s, ct)$

Fig. 1. $\mathsf{KEM} := \mathsf{FO}^{\not\perp}[\mathsf{PKE}, \mathsf{G}, \mathsf{H}]$ for BIKE and SIKE.

validity of c_1. It then retrieves $m' := c_2 \oplus \mathsf{H}(r')$ and checks the validity of the ciphertext by checking $r' = \mathsf{G}(m')$ or not.

SIKE's Decaps performs the test $c_1' = c_1$ but omits the test $c_2' = c_2$. Since Dec retrieves $m' := c_2 \oplus k$ *deterministically*, we do not need to check the equality of c_2 and c_2'.

4 Key-Recovery Plaintext-Checking Attack Against ntrulpr of NTRU Prime

We propose a new key-recovery attack using plaintext-checking oracle against ntrulpr of NTRU Prime [14]. NTRU LPRime (ntrulpr) is a variant of the LPR PKE [54] and has a similar structure to Kyber and Saber. We mimic the KR-PCA against Kyber and Saber proposed by Băetu et al. [10] and Huguenin-Dumittan and Vaudenay [44].

ntrulpr of NTRU Prime: NTRU LPRime has parameter sets $p, q, w, \delta, \tau_0, \tau_1, \tau_2$, and τ_3. We note that $q = 6q' + 1$ for some q' and $q \geq 16w + 2\delta + 3$. For concrete values, see Table 3.

Table 3. Parameter sets of ntrulpr of NTRU Prime

parameter sets	p	q	w	δ	τ_0	τ_1	τ_2	τ_3
`ntrulpr653`	653	4621	252	289	2175	113	2031	290
`ntrulpr761`	761	4591	250	292	2156	114	2007	287
`ntrulpr857`	857	5167	281	329	2433	101	2265	324
`ntrulpr953`	953	6343	345	404	2997	82	2798	400
`ntrulpr1013`	1013	7177	392	450	3367	73	3143	449
`ntrulpr1277`	1277	7879	429	502	3724	66	3469	496

Let $\mathcal{R} := \mathbb{Z}[x]/(x^p - x - 1)$ and $\mathcal{R}_q := \mathbb{Z}_q[x]/(x^p - x - 1)$. Let $\mathcal{S} := \{a = \sum_{i=0}^{p-1} a_i x^i \in \mathcal{R} \mid a_i \in \{-1, 0, +1\}, \mathsf{HW}(a) = w\}$, a set of "short" polynomials.

Table 4. The PCO's behaviors

sk_i	$\mathsf{PCO}(\vec{1}_{256}, ct_0)$	$\mathsf{PCO}(\vec{1}_{256}, ct_1)$
1	1	1
0	1	0
-1	0	0

For $a \in [-(q-1)/2, (q-1)/2]$, define $\mathsf{Round}(a) = 3 \cdot \lceil a/3 \rfloor$.[7] For a polynomial $A = \sum_i a_i x^i \in \mathcal{R}_q$, we define $\mathsf{trunc}(A, l) = (a_0, \ldots, a_{l-1}) \in \mathbb{Z}_q^l$. For $C \in [0, q)$, define $\mathsf{Top}(C) = \lfloor (\tau_1(C + \tau_0) + 2^{14})/2^{15} \rfloor$. For $T \in [0, 16)$, define $\mathsf{Right}(T) = \tau_3 T - \tau_2 \in \mathbb{Z}_q$. For $a \in \mathbb{Z}$, define $\mathsf{Sign}(a) = 1$ if $a < 0$, 0 otherwise.

The underlying CPA-secure PKE scheme[8] works as follows:

– $\mathsf{Gen}(pp)$: Generate $A \leftarrow \mathcal{R}_q$ and $sk \leftarrow \mathcal{S}$. Compute $B := \mathsf{Round}(A \cdot sk)$. Output $pk := (A, B)$ and sk.
– $\mathsf{Enc}(pk, \mu \in \{0,1\}^{256})$: Choose $t \leftarrow \mathcal{S}$ and output

$$(U, V) := \big(\mathsf{Round}(t \cdot A), \mathsf{Top}(\mathsf{trunc}(t \cdot B, 256) + \mu(q-1)/2)\big).$$

– $\mathsf{Dec}(sk, (U, V))$: Compute $r := \mathsf{Right}(V) - \mathsf{trunc}(sk \cdot U, 256) + (4w+1) \cdot \vec{1}_{256} \in \mathbb{Z}^{256}$ and outputs $m := \mathsf{Sign}(r \bmod^{\pm} q)$.

4.1 Key-Recovery Attack

We mainly follow the KR-PCAs against Kyber and Saber in Baetu et al. [10] and Huguenin-Dumittan and Vaudenay [44], but we need some tweaks. Roughly speaking, to determine the i-th coefficient of sk, their attack queries $(a, b \cdot x^i)$ with constant a and b and a candidate plaintext, because in the case of Kyber and Saber, the dimension of V is the same as that of the base ring. However, ntrulpr truncates tB to reduce redundancy, so we need to modify the query ciphertext. Note that we can *shift* the effect of sk_i into *constant coefficient* by multiplying x^{p-i}. That is, for $i = 1, \ldots, p-1$ and $sk = sk_0 + sk_1 x + \ldots sk_{p-1} x^{p-1} \in \mathcal{R}$, we have

$$\begin{aligned} x^{p-i} \cdot sk = {} & sk_i + (sk_i + sk_{i+1})x + (sk_{i+1} + sk_{i+2})x^2 + \cdots + (sk_{p-1} + sk_0)x^{p-i} \\ & + sk_1 x^{p-i+1} + sk_2 x^{p-i+2} + \cdots + sk_{i-1} x^{p-1}. \end{aligned}$$

Using this relation, we show the following two lemmas:

Lemma 4.1 (For general $i \in [1, p)$). *Let $c = \tau_2 - (4w+1)$, $b = \lfloor (c-1)/6 \rfloor \cdot 3$ and $t_\beta = \lfloor (\beta b + c - 1)/\tau_3 \rfloor$ for $\beta \in \{0,1\}$. Let us consider our test ciphertext $ct_\beta = (b \cdot x^{p-i}, (t_\beta, 0, \ldots, 0))$ for $\beta \in \{0,1\}$ and candidate plaintext $\vec{1}_{256}$. Then, we have the relations between the i-th coordinate of decryption key and the behavior of PCO as in Table 4.*

[7] When $q = 6q' + 1$, $\mathsf{Round}([-(q-1)/2, (q-1)/2]) \in [-(q-1)/2, (q-1)/2]$.
[8] 'NTRU LPRime Core' in the specification.

Proof. The decryption algorithm computes $r = \mathsf{Right}\big((t_\beta, 0, \ldots, 0)\big) - \mathsf{trunc}\big(sk \cdot b \cdot x^{p-i}, 256\big) + (4w+1) \cdot \vec{1}_{256}$. Expanding this, we have

$$\begin{cases} r_0 = \tau_3 t_\beta - b \cdot sk_i - c, \\ r_j = -b \cdot (sk_{i+j-1 \bmod p} + sk_{i+j \bmod p}) - c & (j = 1, 2, \ldots, \min\{256, p-i\}) \\ r_j = -b \cdot sk_{j-(p-i) \bmod p} - c & (j = p-i+1, \ldots, \min\{256, p-1\}). \end{cases}$$

Recall that $sk_i \in \{-1, 0, +1\}$ for all i since sk is in $\mathcal{S}$. Thus, we have $r_j \in \{-2b-c, -b-c, -c, b-c, 2b-c\}$ for $j = 1, \ldots, 256$. Since we set $b = \lfloor (c-1)/6 \rfloor \cdot 3 \leq (c-1)/2$, we have $-2b-c > -2c$ and $2b-c < 0$. Fortunately, we have $-2c = -2\tau_2 - 8w - 2 \geq -(q-1)/2$ for all parameter sets. Thus, r_j's are decoded into 1 for $j = 1, \ldots, 256$.

Let us consider r_0. We have

$$r_0 = \tau_3 t_\beta - b \cdot sk_i - c > 0 \iff (\tau_3 t_\beta - c)/b > sk_i$$

By our setting, if $t_\beta = t_0$ (and t_1), then $(\tau_3 t_\beta - c)/b$ is slightly smaller than 0 (and 1) for all parameter sets, respectively. In addition, we have $\tau_3 t_1 + b - c \leq (q-1)/2$ for all parameter sets. Therefore, r_0 for t_0 is decoded into 0 if and only if $sk_i < 0$ and r_0 for t_1 is decoded into 0 if and only if $sk_i < 1$. This completes the proof. □

By a similar argument, we have the following lemma on sk_0.

Lemma 4.2 ($i = 0$). *Let $c = \tau_2 - (4w+1)$, $b = \lceil (c-1)/6 \rceil \cdot 3$ and $t_\beta = \lfloor (\beta b + c - 1)/\tau_3 \rfloor$ for $\beta \in \{0, 1\}$. Let us consider our test ciphertext $ct_\beta = (b, (t_\beta, 0, \ldots, 0))$ and candidate plaintext $\vec{1}_{256}$. Then, we have the relations between the constant term of decryption key and the behavior of PCO as in Table 4.*

Using the above lemmas, we can determine sk_i for $i = 0, \ldots, p-1$ by testing $2p$ queries with the PCO.

4.2 Trade-Off

We observe that an adversary can obtain $K' = \mathsf{H}(m', ct)$ by skipping the equality test instead of the equality of K' and K_{guess} or the equality of m' and m_{guess}. Therefore, the adversary can check if $m' = m_{\mathsf{guess}}$ or not by computing $K_{\mathsf{guess}} = \mathsf{H}(m_{\mathsf{guess}}, ct)$ by itself. This enables the adversary to determine ℓ coefficients of the secret key at once by sacrificing the computational efficiency.

For simplicity, we let $\ell = 2^k < 256$.

Determine $sk_{y\ell}, \ldots, sk_{y\ell+\ell-1}$ for $y = 0, \ldots, 256/\ell - 1$: Let us determine ℓ coefficients $sk_{y\ell}, \ldots, sk_{y\ell+\ell-1}$ of sk at once, where $y = 0, \ldots, 256/\ell - 1$. Suppose that we query two ciphertexts

$$ct_\beta = (U, V_\beta) = \big(b, (\overbrace{0, \ldots, 0}^{y\ell}, \overbrace{t_\beta, \ldots, t_\beta}^{\ell}, \overbrace{0, \ldots, 0}^{256-(y+1)\ell}))$$

for $\beta \in \{0,1\}$. The decryption algorithm computes $r = \mathsf{Right}(V_\beta) - \mathsf{trunc}(sk \cdot b, 256) + (4w+1) \cdot \vec{1}_{256}$. Expanding this, we have

$$r_j = \begin{cases} \tau_3 t_\beta - b \cdot sk_j - c & (j = y\ell, \dots, y\ell + \ell - 1) \\ -b \cdot sk_j - c & (j = 0, \dots, y\ell - 1, (y+1)\ell, \dots, 256). \end{cases}$$

By using the argument in the proof of Lemma 4.1, r_j's are decoded into 1 for $j = 0, \dots, y\ell - 1, (y+1)\ell, \dots, 256$. We also have, for $j = y\ell, \dots, y\ell + \ell - 1$, r_j for t_1 is decoded into 0 if and only if $sk_i < 0$ and r_j for t_2 is decoded into 0 if and only if $sk_i < 1$.

Seeing $K = \mathsf{H}(m', ct_\beta)$ where $m' = \mathsf{Dec}(sk, ct_\beta)$, we compute $K_{\mathsf{guess}} = \mathsf{H}(m_{\mathsf{guess}}, ct_\beta)$ for $m_{\mathsf{guess}} = \vec{1}_{y\ell} \| m'' \| \vec{1}_{256-(y+1)\ell}$ for all $m'' \in \{0,1\}^\ell$ and determine sk_j for $j = y\ell, \dots, y\ell + \ell - 1$.

Determine $sk_{y\ell}, \dots, sk_{y\ell+\ell-1}$ for $y = 256/\ell, \dots, \lfloor p/\ell \rfloor$*:* Suppose that we have determined $y\ell$ coefficients $sk_0, \dots, sk_{y\ell-1}$ for some $y \in \{256/\ell, \dots, \lfloor p/\ell \rfloor\}$. Let us determine ℓ coefficients $sk_{y\ell}, \dots, sk_{y\ell+\ell-1}$ at once: Let $t_\beta = \lfloor (\beta b + c - 1)/\tau_3 \rfloor$ for $\beta \in \{-1, 0, 1, 2\}$. Suppose that we query four ciphertexts

$$ct_\beta = (U, V_\beta) = \big(b \cdot x^{p - y\ell - 1}, (0, \overbrace{t_\beta, \dots, t_\beta}^{\ell}, \overbrace{0, \dots, 0}^{256-\ell-1})\big)$$

for $\beta \in \{-1, 0, 1, 2\}$. The decryption algorithm computes $r = \mathsf{Right}(V_\beta) - \mathsf{trunc}(sk \cdot bx^{p-y\ell-1}, 256) + (4w+1) \cdot \vec{1}_{256}$. Expanding this, we have

$$r_j = \begin{cases} -b \cdot sk_{y\ell-1} - c & (j = 0) \\ \tau_3 t_\beta - b \cdot (sk_{y\ell+j-2 \bmod p} + sk_{y\ell+j-1 \bmod p}) - c & (j = 1, 2, \dots, \ell) \\ -b \cdot (sk_{y\ell+j-2 \bmod p} + sk_{y\ell+j-1 \bmod p}) - c & (j = \ell+1, \dots, \min\{256, p - y\ell\}) \\ -b \cdot sk_{j-(p-i) \bmod p} - c & (j = \min\{256, p - y\ell + 1\}, \dots, 256). \end{cases}$$

By using the argument in the proof of Lemma 4.1, r_j's are decoded into 1 for $j = 0$ and $j = \ell + 1, \dots, 256$.

Let us consider r_j for $j = 1, \dots, \ell$. We have

$$r_j = \tau_3 t_\beta - b \cdot (sk_j + sk_{j+1}) - c > 0 \iff (\tau_3 t_\beta - c)/b > sk_j + sk_{j+1}.$$

By our setting, $(\tau_3 t_\beta - c)/b$ is slightly smaller than β for all parameter sets, respectively. In addition, we have $-(q-1)/2 \le \tau_3 t_\beta - 2b - c$ and $\tau_3 t_\beta + 2b - c \le (q-1)/2$ for all parameter sets. Therefore, r_j for t_β is decoded into 0 if and only if $sk_i < \beta$.

Seeing $K' = \mathsf{H}(m', ct_\beta)$ where $m' = \mathsf{Dec}(sk, ct_\beta)$, we compute $K_{\mathsf{guess}} = \mathsf{H}(m_{\mathsf{guess}}, ct_\beta)$ for $m_{\mathsf{guess}} = 1 \| m'' \| \vec{1}_{256-\ell-1}$ for all $m'' \in \{0,1\}^\ell$ and determine $sk_{y\ell+j-2} + sk_{y\ell+j-1} \in \{-2, -1, 0, 1, 2\}$ for $j = 1, \dots, \ell$. Since we know $sk_{y\ell-1}$, we can determine $sk_{y\ell}, \dots, sk_{y\ell+\ell-1}$ sequentially.

5 Skipping the Equality Test by Skipping a Single Instruction

In this section, we describe the fault-injection attack on the equality checking of each KEM implementation. First, we examine the implementation of pqm4 [49] for each scheme and discuss the possibility of skipping the equivalence test. To identify the instructions to be skipped, we cross-compiled the C code in pqm4 with GCC 8.3.1 running on Debian bullseye. The compilation options were basically according to pqm4, with "-O3" as an optimization option.

We do not mention the attacks on Classic McEliece and HQC in this section because pqm4 does not include their ARM Cortex M4-optimized code. Hereafter, we describe the possibility of skip attacks on NTRU Prime, FrodoKEM, Kyber, Saber, NTRU, BIKE, and SIKE.

If the reader is unfamiliar to Arm Cortex M4, please see the manual[9].

5.1 NTRU Prime – CCA Bug

The functions in the C code related to the FO-like transformation are `crypto_kem_dec`, `Decap`, and `Ciphertexts_diff_mask`.[10] Figure 2 shows the source code of NTRU Prime's comparison in pqm4. Note that we omit the `crypto_kem_dec` function as it just calls `Decap`.

Let us consider how `Ciphertexts_diff_mask` computes the return value. It initializes the `uint16` variable `differentbits` as `0`. After some computations, it outputs `((-1)-((differentbits-1)>>31))` in line 17. The value is initialized as `0` and *unchanged* before the return value is computed; these computations only involve `differentbits2`. Thus, we eventually obtain `0` as the result of `(-1)-((0-1)>>31)` and `ciphertexts_diff_mask` always outputs `0`.

`Decap` first decrypts $r := \mathsf{Dec}(sk, c)$ in line 13 and encodes it into `r_enc` and re-encrypts it into `cnew` in line 14. In line 15, `mask` is always `0`, since `Ciphertexts_diff_mask` always returns `0` as we explained. Thus, `r_enc`, which is the result of faulty decryption, is unchanged, and `Decap` always sets `k` as the result of `H(1,r_enc,c)`. This means that there is no re-encryption check and the implementation opens the attack surface of chosen-ciphertext attacks.

5.2 FrodoKEM – Timing Attack

The decapsulation of FrodoKEM is performed in the `crypto_kem_dec` function.[11] Figure 3 shows the source code of the equality test in the function. From the

[9] https://developer.arm.com/documentation/100166/0001. See https://developer.arm.com/documentation/100166/0001/Programmers-Model/Instruction-set-summary/Table-of-processor-instructions?lang=en for instruction set.

[10] The source code of these functions is https://github.com/mupq/pqm4/blob/master/crypto_kem/sntrup761/m4f/kem.c.

[11] https://github.com/mupq/pqm4/blob/master/crypto_kem/frodokem640shake/m4/kem.c.

```c
static int Ciphertexts_diff_mask(const unsigned char *c,
                                 const unsigned char *c2)
{
  uint16 differentbits = 0;
  int len = Ciphertexts_bytes+Confirm_bytes;

  int *cc = (int *)(void *)c;
  int *cc2 = (int *)(void *)c2;
  int differentbits2 = 0;
  for (len-=4 ;len>=0; len-=4) {
    differentbits2 = __USADA8((*cc++),(*cc2++),differentbits2);
  }
  c = (unsigned char *)(void *) cc;
  c2 = (unsigned char *)(void *) cc2;
  for (len &= 3; len > 0; len--)
    differentbits2 =__USADA8((*c++),(*c2++),differentbits2);
  return ((-1)-((differentbits-1)>>31));
}
```

```c
static void Decap(unsigned char *k,const unsigned char *c,
                                 const unsigned char *sk)
{
  const unsigned char *pk = sk + SecretKeys_bytes;
  const unsigned char *rho = pk + PublicKeys_bytes;
  const unsigned char *cache = rho + Inputs_bytes;
  Inputs r;
  unsigned char r_enc[Inputs_bytes];
  unsigned char cnew[Ciphertexts_bytes+Confirm_bytes];
  int mask;
  int i;

  ZDecrypt(r,c,sk);
  Hide(cnew,r_enc,r,pk,cache);
  mask = Ciphertexts_diff_mask(c,cnew);
  for (i = 0;i < Inputs_bytes;++i)
    r_enc[i] ^= mask&(r_enc[i]^rho[i]);
  HashSession(k,1+mask,r_enc,c);
}
```

Fig. 2. NTRU Prime's comparison in pqm4.

source code, this function uses the `memcmp` function with `&&` to compare the ciphertext and the re-encryption result. This indicates that the current implementation is still vulnerable to the timing attack by Guo et al. [38].

```
// Is (Bp == BBp & C == CC) = true
if (memcmp(Bp, BBp, 2 * PARAMS_N * PARAMS_NBAR) == 0 &&
      memcmp(C, CC, 2 * PARAMS_NBAR * PARAMS_NBAR) == 0) {
    // Load k' to do ss = F(ct || k')
    memcpy(Fin_k, kprime, CRYPTO_BYTES);
} else {
    // Load s to do ss = F(ct || s)
    memcpy(Fin_k, sk_s, CRYPTO_BYTES);
}
shake(ss, CRYPTO_BYTES, Fin, CRYPTO_CIPHERTEXTBYTES
                                    + CRYPTO_BYTES);
```

Fig. 3. FrodoKEM's comparison in pqm4

5.3 Kyber, Saber, and NTRU – cmov

In this subsection, we describe the skip attacks on Kyber, Saber, and NTRU among the finalists. The basic idea of the skip attacks on these implementations is identical, and thus we describe the case of Saber as an example to explain the skip attack procedure. Figure 4 shows the `crypto_kem_dec` function that performs the decapsulation of FO transformation[12].

The `crypto_kem_dec` function performs re-encryption using the `indcpa_kem_enc_cmp` function at line 14 and stores the comparison results of the ciphertext and the re-encryption result into a variable `fail`. If these ciphertexts are not the same, `fail` becomes `1` and, if they are the same, `fail` becomes `0`. At line 16, the `cmov` substitutes a random value for `kr` when `fail` is `1`. Note here that the hash value calculated from the decrypted result is stored in the variable `kr` before `cmov` is called, and this means that we can perform a key-recovery attack by skipping the call of `cmov` even when `fail` is `1`.

Figure 5 shows the assembly code corresponding to the call of `cmov`. This program first calls the `sha3_256` function at line 1, prepares the arguments of `cmov` at line 4–7, calls `cmov` at line 8, and finally prepares the arguments and call the `sha3_256` function at line 10–14. From this code, we notice that Saber can be attacked by skipping `bl cmov` at line 8 using fault injection. In addition to Saber, NTRU and Kyber also use `cmov` in the same manner, and therefore, this attack can be applied to all of them.

[12] https://github.com/mupq/pqm4/blob/master/crypto_kem/saber/m4f/kem.c.

```
    int crypto_kem_dec(uint8_t *k, const uint8_t *c,
                                        const uint8_t *sk)
{
    uint8_t fail;
    uint8_t buf[64];
    uint8_t kr[64]; // Will contain key, coins
    const uint8_t *pk = sk + SABER_INDCPA_SECRETKEYBYTES;
    const uint8_t *hpk = sk + SABER_SECRETKEYBYTES - 64;
                       // Save hash by storing h(pk) in sk

    indcpa_kem_dec(sk, c, buf);
    memcpy(buf + 32, hpk, 32);
    sha3_512(kr, buf, 64);
    fail = indcpa_kem_enc_cmp(buf, kr + 32, pk, c);
    sha3_256(kr + 32, c, SABER_BYTES_CCA_DEC);
    cmov(kr, sk + SABER_SECRETKEYBYTES - SABER_KEYBYTES,
                                    SABER_KEYBYTES, fail);
    sha3_256(k, kr, 64);
    return (0);
}
```

Fig. 4. Saber's comparison in pqm4

```
    bl  sha3_256
.LVL26:
.loc 1 79 3 is_stmt 1 view .LVU62
    uxtb    r3, r7
    add r1, r4, #1536
    add r0, sp, #64
    movs    r2, #32
    bl  cmov
.LVL27:
    .loc 1 82 3 view .LVU63
    movs    r2, #64
    mov r0, r6
    add r1, sp, r2
    bl  sha3_256
```

Fig. 5. Assembly code of Saber's comparison in pqm4

5.4 BIKE – for Loop

We describe the skip attack on BIKE in this subsection. Figure 6 shows the C code of BIKE's comparison in the decapsulation[13]. We also show `secure_cmp` function and `secure_l32_mask` function in Fig. 7. Line 4–7 in Fig. 6 compares

[13] https://github.com/mupq/pqm4/blob/master/crypto_kem/bikel1/m4f/kem.c.

the hash value of the original message and that of the decrypted message from the ciphertext. Then, if they are equal, the `for` block at line 12–15 stores the decrypted message into `m_prime.raw[i]`. Therefore, the goal of the fault-injection attack is to store the decrypted message even when these hash values differ. For this purpose, we need to force the variable `mask` to be `0`.

Figure 8 shows the assembly code corresponding to the line 6–11 in the C code to explain the position of a fault injection. Line 1–30 and line 31–44 in the assembly code correspond to line 6 and line 11 in the C code, respectively. The operation we need to skip for a key-recovery attack is "`ldr r2, [sp, #20]`" at line 33 in this assembly code for the following reason.

Before line 33 in the assembly code, the `r2` register is used in "`cmp r2, #0`" at line 26. This corresponds to "`return (0 == res);`" at line 11 in `secure_cmp` function (Fig. 7). Therefore, at this time, the `r2` register contains the value of the `res` variable. The value of the `r2` register does not become `0` from the attack assumption because the value of the `res` variable is not `0` when the two arguments of `secure_cmp` are not equal. Thus, the `r2` register must be a non-zero value if line 33 in the assembly code is skipped. After line 33, the value of the `r2` register is used at line 41. This line corresponds to line 9 in the `secure_l32_mask` function (Fig. 7). The `secure_l32_mask` function compares the two arguments `v1` and `v2` and returns `0` when `v1 < v2` holds. `mask` becomes `0` when `v2` is not `0` because `v1` is `0` as shown in Fig. 6. Meanwhile, we note that the variable `v2` does not become `0` when we skip line 33 in the assembly code because the `r2` register corresponds to the `v2` variable. From the above, we can fix `mask` to `0` by the fault injection, and thus the key-recovery attack is possible.

5.5 SIKE – Simple if

This subsection describe the skip attack on SIKE. Figure 9 and Fig. 10 shows the C code and its assembly of the comparison process in the FO transformation[14].

The target of fault injection in C code is the `if` statement at line 4–6. The assembly code in Fig. 10 corresponds to the `if` statement. The process of condition in the `if` statement at line 4 in the C code corresponds to line 1–3 in the assembly code. In the assembly code, "`bl memcmp`" compares the variables `c0_` and `ct`. If they differ, "`cbnz r0, .L500`" performs a jump to line 23. Note that, even if we jump to line 23, the procedure comes back to line 4 because of "`b .L495`" at line 33. In other words, line 23–33 in the assembly code correspond to the process in the `if` block at line 5 in the C code. Thus, we can perform the skip attack on SIKE by injecting a fault into "`cbnz r0, .L500`" at line 3.

[14] https://github.com/mupq/pqm4/blob/master/crypto_kem/sikep434/m4/sike.inc.

```
// Check if H(m') is equal to (e0', e1')
// (in constant-time)
GUARD(function_h(&e_tmp, &m_prime));
success_cond = secure_cmp(PE0_RAW(&e_prime),
                             PE0_RAW(&e_tmp), R_BYTES);
success_cond &= secure_cmp(PE1_RAW(&e_prime),
                             PE1_RAW(&e_tmp), R_BYTES);

// Compute either K(m', C) or K(sigma, C) based on the
//                                    success condition
mask = secure_l32_mask(0, success_cond);
for(size_t i = 0; i < M_BYTES; i++) {
  m_prime.raw[i] &= u8_barrier(~mask);
  m_prime.raw[i] |= (u8_barrier(mask) & l_sk.sigma.raw[i]);
}
```

Fig. 6. BIKE's comparison in pqm4.

```
_INLINE_ uint32_t secure_cmp(IN const uint8_t *a,
                             IN const uint8_t *b,
                             IN const uint32_t size)
{
  volatile uint8_t res = 0;

  for(uint32_t i = 0; i < size; ++i) {
    res |= (a[i] ^ b[i]);
  }

  return (0 == res);
}
```

```
// Return 0 if v1 < v2, (-1) otherwise
_INLINE_ uint32_t secure_l32_mask(IN const uint32_t v1,
                                  IN const uint32_t v2)
{
  // If v1 >= v2 then the subtraction result is 0^32||(v1-v2).
  // else it is 1^32||(v2-v1+1).
  // Subsequently, negating the upper
  // 32 bits gives 0 if v1 < v2 and otherwise (-1).
  return ~((uint32_t)(((uint64_t)v1 - (uint64_t)v2) >> 32));
}
```

Fig. 7. secure_cmp and secure_l32_mask function of BIKE in pqm4.

```
.L26:
    .loc 1 69 5 is_stmt 1 view .LVU627
    ldrb    r2, [r5, #1]!
.LVL169:
    .loc 1 69 9 view .LVU629
    ldrb    r4, [r1, #1]!
    ldrb    r3, [sp, #18]
    eors    r2, r2, r4
    orrs    r3, r3, r2
    .loc 1 68 3 view .LVU630
    cmp     r0, r5
    .loc 1 69 9 view .LVU631
    strb    r3, [sp, #18]
.LVL170:
    .loc 1 68 3 view .LVU632
    bne     .L26
.LBE629:
    .loc 1 72 3 is_stmt 1 view .LVU633
.LVL171:
    .loc 1 72 13 is_stmt 0 view .LVU634
    ldrb    r2, [sp, #18]
.LBE628:
.LBE627:
    .loc 2 278 16 view .LVU635
    ldr     r3, [sp, #20]
    cmp     r2, #0
    ite     ne
    movne    r3, #0
    andeq    r3, r3, #1
    str     r3, [sp, #20]
    .loc 2 282 3 is_stmt 1 view .LVU636
    .loc 2 282 10 is_stmt 0 view .LVU637
    ldr     r2, [sp, #20]
.LVL172:
.LBB630:
.LBI630:
    .loc 1 113 19 is_stmt 1 view .LVU638
.LBB631:
    .loc 1 140 3 view .LVU639
    .loc 1 140 37 is_stmt 0 view .LVU640
    rsbs    r2, r2, #0
    sbc     r3, r3, r3
    .loc 1 140 10 view .LVU641
    mvns    r5, r3
```

Fig. 8. Assembly code of BIKE's comparison in pqm4

```
// Generate shared secret ss <- H(m||ct)
//                            or output ss <- H(s||ct)
EphemeralKeyGeneration_A(ephemeralsk_, c0_);
if (memcmp(c0_, ct, CRYPTO_PUBLICKEYBYTES) != 0) {
    memcpy(temp, sk, MSG_BYTES);
}
memcpy(&temp[MSG_BYTES], ct, CRYPTO_CIPHERTEXTBYTES);
shake256(ss, CRYPTO_BYTES, temp,
                CRYPTO_CIPHERTEXTBYTES+MSG_BYTES);
```

Fig. 9. SIKE's comparison in pqm4

```
    bl  memcmp
    .loc 5 88 8 view .LVU4945
    cbnz    r0, .L500
.L495:
    .loc 5 91 5 is_stmt 1 view .LVU4946
    mov r1, r4
    add r0, sp, #508
    mov r2, #346
    bl  memcpy
    .loc 5 92 5 view .LVU4947
    mov r0, r8
    add r2, sp, #492
    mov r3, #362
    movs    r1, #16
    bl  shake256
    .loc 5 94 5 view .LVU4948
    .loc 5 95 1 is_stmt 0 view .LVU4949
    movs    r0, #0
    add sp, sp, #856
    .cfi_remember_state
    .cfi_def_cfa_offset 24
    pop {r4, r5, r6, r7, r8, pc}
.L500:
    .cfi_restore_state
    .loc 5 89 9 is_stmt 1 view .LVU4950
    ldr r0, [r5]
    ldr r1, [r5, #4]
    ldr r2, [r5, #8]
    ldr r3, [r5, #12]
    add r5, sp, #492
    .loc 5 89 9 is_stmt 0 view .LVU4951
    stmia   r5!, {r0, r1, r2, r3}
    b   .L495
```

Fig. 10. Assembly code of SIKE's comparison in pqm4

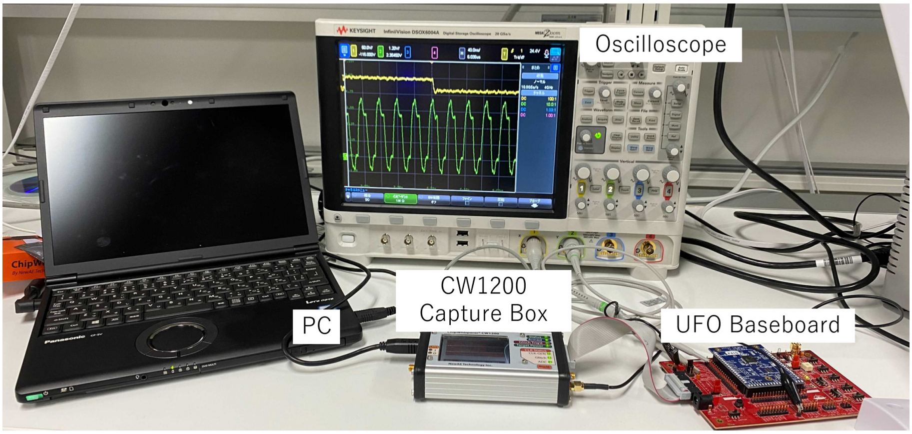

Fig. 11. Experimental setup overview.

Table 5. Numbers of failures and successes when we conducted 100 skip attacks on each scheme

Name	# failures	# Successes	# required queries	Expected time [s]
Kyber – `Kyber512`	60	52	5908	626
NTRU – `ntruhps2048509`	74	46	2235	384
Saber – `LightSaber`	33	33	15515	1,567
BIKE – `Bikel1`	49	34	-	-
SIKE – `sikep434`	30	15	1787	19,478

6 Experimental Attacks

In this section, we conduct the experimental skip attacks on the pqm4 implementation of the above mentioned KEM schemes. The target schemes in this section are Kyber, NTRU, Saber, BIKE, and SIKE, which were shown to be attackable by a single fault injection in the previous section. In this experiment, we used the parameters of the security level 1 for all schemes.

6.1 Setup

Figure 11 shows the experimental environment. The target chip under attack is an STM32F415 microcontroller with an ARM Cortex M4 core, which is a de-facto standard platform to evaluate software implementation of schemes running in NIST's PQC process. The target device is mounted on a ChipWhisperer cw308 UFO baseboard, which enables us to perform fault-injection attacks using a glitchy clock. The ChipWhisperer cw1200 capture box is used to generate the base clock, and the clock frequency was set to 24 MHz. The glitch parameters for instruction skipping were searched by sweeping the parameters to find the one that successfully skips the instruction. We use the implementation in pqm4 for each KEM scheme, and "O3" was specified as an optimization option during compilation.

6.2 Results

Table 5 reports the experimental results of the proposed skip attacks. In Table 5, we show the number of times when a fault occurred on the device and the number of successful instruction skips when we performed 100 fault injections for each scheme. Also, the table shows the number of required queries to recover the secret key from each scheme using fault injection.[15,16] These required query numbers are calculated by multiplying the minimum required number of queries for a key-recovery attack and the inverse of the success rate of a skip attack. We only omitted the number of required queries for the case of BIKE in this table because it is difficult to fully recover the secret key. We also show the expected time to recover the secret key for each scheme. From the table, we confirm that the probability of a successful attack was about 15–50%, and there is a difference in the probability of successful attacks among Saber, Kyber, and NTRU, although the fault-injection capability is almost the same. This would be because of the difference in instructions before and after the call of the `cmov` function that affects the state of pipeline registers in the microcontroller.

In addition, in this experiment, the injected faults did not always cause a single instruction skip as expected and sometimes crashed the device, which led to a non-negligible cost for an oracle access. A similar phenomenon was also observed in [60] in fault-injection attacks on lattice KEMs using ChipWhisperer, and more sophisticated equipment for fault injection should achieve higher attack stability.

7 Countermeasure

Default fail: The one of major countermeasures is the 'default fail' technique, which initiates a variable with the fail result and if a condition is satisfied then the variable is overwritten by the sensitive data [32].

Recall Saber's decapsulation in Fig. 4: We want to compute $K = \mathsf{H}(\bar{K}', \mathsf{H}(ct))$ or $\mathsf{H}(s, \mathsf{H}(ct))$ depending on the re-encryption test result, where $\bar{K}'$ is computed from the decrypted result m' and pk and s is a secret seed. If we skip the function call of `cmov`, then $\bar{K}'$ in `kr` is unchanged and we obtain $K = \mathsf{H}(\bar{K}', \mathsf{H}(ct))$ as the faulty decapsulation result. According to the 'default fail' technique, we put a secret seed s as the *default* value of `kr` and apply `cmov` to overwrite s by $\bar{K}'$

[15] In practice, we may need more queries than the values shown in the table, because the value of the secret key may occasionally carry an error due to an inserted fault. For simplicity, we ignore such situations here.

[16] On Saber and Kyber, we have trade-offs between the number of expected queries and efficiency. In this table, we use $\ell = 1$.

depending on the value `flag`. (In addition, we will need to clear the original $\bar{K}'$.) If it was, then skipping `cmov` results in $K = \mathsf{H}(s, \mathsf{H}(ct))$ irrelevant to the decrypted result m'.

Moreover, a concrete assembly-level implementation of conditional branch resistant to single instruction skipping by default fail was presented in [32]. Their countermeasure enables that sensitive instruction(s) should be performed only if a condition is surely tested and satisfied. In other words, if the condition test is skipped by a single-fault attack, the implementation with their countermeasure always outputs the rejection.

Instruction duplication: The other major countermeasures is the 'assembly-level instruction duplication' technique: If every instructions are duplicated carefully, then a single-fault instruction skipping attack is ineffective. See, e.g., [12] for the effectiveness and cost.

Random delay: Random delays are yet another major countermeasure of fault-injection analysis. If a random delay is inserted, then it is hard to determine the timing for injecting a fault. See, e.g., [24] for such technique.

8 Conclusion

From the viewpoint of fault-injection attacks, we have investigate *all* NIST PQC Round 3 KEM candidates, which use variants of the FO transformation. We survey effective key-recovery attacks if we can skip the equality test.

We found the existing key-recovery attacks against Kyber, NTRU, Saber, FrodoKEM, HQC, and SIKE (Table 2). We have proposed a new key-recovery attack against ntrulpr of NTRU Prime. We also pointed out trade-offs between the number of queries and computational costs when the target is Kyber, Saber, or ntrulpr. We also reported attacks against sntrupr of NTRU Prime and BIKE that lead to leakage of information of secret keys.

The open-source pqm4 library contains Kyber, NTRU, Saber, BIKE, FrodoKEM, NTRU Prime, and SIKE. We show that giving a single instruction-skipping fault in the decapsulation processes leads to skipping the equality test *virtually* for Kyber, NTRU, Saber, BIKE, and SIKE. We also report the implementation of NTRU Prime allows chosen-ciphertext attacks freely and the timing side-channel of FrodoKEM reported in Guo et al. [38] remains.

Finally, we have reported the experimental attacks against Kyber, NTRU, Saber, BIKE, and SIKE on pqm4. We also discuss possible countermeasures.

Acknowledgment. The authors would like to thank to anonymous reviewers of Asiacrypt 2021 for their helpful and insightful comments.

References

1. ISO/IEC 18033-2:2006 information technology – security techniques – encryption algorithms – part 2: asymmetric ciphers (2006). https://www.iso.org/standard/37971.html

2. Abdalla, M., Benhamouda, F., Pointcheval, D.: Public-key encryption indistinguishable under plaintext-checkable attacks. In: Katz, J. (ed.) PKC 2015. LNCS, vol. 9020, pp. 332–352. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-46447-2_15
3. Aggarwal, D., Maurer, U.: Breaking RSA generically is equivalent to factoring. In: Joux, A. (ed.) EUROCRYPT 2009. LNCS, vol. 5479, pp. 36–53. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-01001-9_2
4. Aguilar Melchor, C., et al.: HQC. Technical report, National Institute of Standards and Technology (2020)
5. Ajtai, M., Dwork, C.: A public-key cryptosystem with worst-case/average-case equivalence. In: STOC 1997, pp. 284–293. ACM Press, May 1997
6. Alagic, G., et al.: NISTIR 8309: status report on the second round of the NIST post-quantum cryptography standardization process, July 2020
7. Albrecht, M.R., et al.: Classic McEliece. Technical report, National Institute of Standards and Technology (2020)
8. Antipa, A., Brown, D., Menezes, A., Struik, R., Vanstone, S.: Validation of elliptic curve public keys. In: Desmedt, Y.G. (ed.) PKC 2003. LNCS, vol. 2567, pp. 211–223. Springer, Heidelberg (2003). https://doi.org/10.1007/3-540-36288-6_16
9. Aragon, N., et al.: BIKE. Technical report, National Institute of Standards and Technology (2020)
10. Băetu, C., Durak, F.B., Huguenin-Dumittan, L., Talayhan, A., Vaudenay, S.: Misuse attacks on post-quantum cryptosystems. In: Ishai, Y., Rijmen, V. (eds.) EUROCRYPT 2019, Part II. LNCS, vol. 11477, pp. 747–776. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-17656-3_26
11. Barenghi, A., Bertoni, G., Perrinello, E., Pelosi, G.: Low voltage fault attacks on the RSA cryptosystem. In: FDTC 2009. IEEE Computer Society (2009)
12. Barenghi, A., Breveglieri, L., Koren, I., Pelosi, G., Regazzoni, F.: Countermeasures against fault attacks on software implemented AES: effectiveness and cost. In: WESS 2010 (2010)
13. Bellare, M. (ed.): CRYPTO 2000, LNCS, vol. 1880. Springer, Heidelberg, August 2000. https://doi.org/10.1007/3-540-44598-6
14. Bernstein, D.J., et al.: NTRU Prime. Technical report, National Institute of Standards and Technology (2020)
15. Biehl, I., Meyer, B., Müller, V.: Differential fault attacks on elliptic curve cryptosystems. In: Bellare [13], pp. 131–146 (2000)
16. Biham, E., Shamir, A.: Differential fault analysis of secret key cryptosystems. In: Kaliski, B.S. (ed.) CRYPTO 1997. LNCS, vol. 1294, pp. 513–525. Springer, Heidelberg (1997). https://doi.org/10.1007/BFb0052259
17. Bindel, N., Hamburg, M., Hövelmanns, K., Hülsing, A., Persichetti, E.: Tighter proofs of CCA security in the quantum random oracle model. In: Hofheinz, D., Rosen, A. (eds.) TCC 2019, Part II. LNCS, vol. 11892, pp. 61–90. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-36033-7_3
18. Blömer, J., Günther, P.: Singular curve point decompression attack. In: FDTC 2015, pp. 71–84. IEEE Computer Society (2015)
19. Boneh, D., Dagdelen, Ö., Fischlin, M., Lehmann, A., Schaffner, C., Zhandry, M.: Random oracles in a quantum world. In: Lee, D.H., Wang, X. (eds.) ASIACRYPT 2011. LNCS, vol. 7073, pp. 41–69. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-25385-0_3
20. Boneh, D., DeMillo, R.A., Lipton, R.J.: On the importance of eliminating errors in cryptographic computations. J. Cryptol. **14**(2), 101–119 (2001)

21. Boneh, D., Venkatesan, R.: Breaking RSA may not be equivalent to factoring. In: Nyberg, K. (ed.) EUROCRYPT 1998. LNCS, vol. 1403, pp. 59–71. Springer, Heidelberg (1998). https://doi.org/10.1007/BFb0054117
22. Chen, C., et al.: NTRU. Technical report, National Institute of Standards and Technology (2020)
23. Cheon, J.H., Takagi, T. (eds.): ASIACRYPT 2016, Part I. LNCS, vol. 10031. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-53887-6
24. Coron, J.-S., Kizhvatov, I.: An efficient method for random delay generation in embedded software. In: Clavier, C., Gaj, K. (eds.) CHES 2009. LNCS, vol. 5747, pp. 156–170. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-04138-9_12
25. Costello, C.: The case for SIKE: a decade of the supersingular isogeny problem. Cryptology ePrint Archive, Report 2021/543 (2021). https://eprint.iacr.org/2021/543
26. Cramer, R., Shoup, V.: Design and analysis of practical public-key encryption schemes secure against adaptive chosen ciphertext attack. SIAM J. Comput. **33**(1), 167–226 (2003)
27. D'Anvers, J.P., et al.: SABER. Technical report, National Institute of Standards and Technology (2020)
28. De Feo, L., Jao, D., Plût, J.: Towards quantum-resistant cryptosystems from supersingular elliptic curve isogenies. J. Math. Cryptol. **8**(3), 209–247 (2014)
29. Dent, A.W.: A designer's guide to KEMs. In: Paterson, K.G. (ed.) Cryptography and Coding 2003. LNCS, vol. 2898, pp. 133–151. Springer, Heidelberg (2003). https://doi.org/10.1007/978-3-540-40974-8_12
30. Diffie, W., Hellman, M.E.: New directions in cryptography. IEEE Trans. Inf. Theory **22**(6), 644–654 (1976)
31. Ding, J., Deaton, J., Schmidt, K., Vishakha, Zhang, Z.: A simple and practical key reuse attack on NTRU cryptosystem. Cryptology ePrint Archive, Report 2019/1022 (2019). https://eprint.iacr.org/2019/1022
32. Endo, S., Homma, N., Hayashi, Y., Takahashi, J., Fuji, H., Aoki, T.: A multiple-fault injection attack by adaptive timing control under black-box conditions and a countermeasure. In: Prouff, E. (ed.) COSADE 2014. LNCS, vol. 8622, pp. 214–228. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-10175-0_15
33. Endo, S., Sugawara, T., Homma, N., Aoki, T., Satoh, A.: An on-chip glitchy-clock generator for testing fault injection attacks. J. Crypt. Eng. **1**(4), 265–270 (2011)
34. Fluhrer, S.: Cryptanalysis of ring-LWE based key exchange with key share reuse. Cryptology ePrint Archive, Report 2016/085 (2016). https://eprint.iacr.org/2016/085
35. Fujisaki, E., Okamoto, T.: Secure integration of asymmetric and symmetric encryption schemes. In: Wiener [78], pp. 537–554 (1999)
36. Fujisaki, E., Okamoto, T.: Secure integration of asymmetric and symmetric encryption schemes. J. Cryptol. **26**(1), 80–101 (2013)
37. Galbraith, S.D., Petit, C., Shani, B., Ti, Y.B.: On the security of supersingular isogeny cryptosystems. In: Cheon and Takagi [23], pp. 63–91 (2016)
38. 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. In: Micciancio, D., Ristenpart, T. (eds.) CRYPTO 2020, Part II. LNCS, vol. 12171, pp. 359–386. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-56880-1_13
39. Guo, Q., Johansson, T., Stankovski, P.: A key recovery attack on MDPC with CCA security using decoding errors. In: Cheon and Takagi [23], pp. 789–815 (2016)

40. Hall, C., Goldberg, I., Schneier, B.: Reaction attacks against several public-key cryptosystem. In: Varadharajan, V., Mu, Y. (eds.) ICICS 1999. LNCS, vol. 1726, pp. 2–12. Springer, Heidelberg (1999). https://doi.org/10.1007/978-3-540-47942-0_2
41. Hayashi, Y., Homma, N., Sugawara, T., Mizuki, T., Aoki, T., Sone, H.: Non-invasive trigger-free fault injection method based on intentional electromagnetic interference. In: Proceedings of The Non-Invasive Attack Testing Workshop - NIAT 2011, September 2011
42. Hofheinz, D., Hövelmanns, K., Kiltz, E.: A modular analysis of the Fujisaki-Okamoto transformation. In: Kalai, Y., Reyzin, L. (eds.) TCC 2017, Part I. LNCS, vol. 10677, pp. 341–371. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-70500-2_12
43. Howe, J., Prest, T., Apon, D.: SoK: how (not) to design and implement post-quantum cryptography. In: Paterson, K.G. (ed.) CT-RSA 2021. LNCS, vol. 12704, pp. 444–477. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-75539-3_19
44. Huguenin-Dumittan, L., Vaudenay, S.: Classical misuse attacks on NIST round 2 PQC. In: Conti, M., Zhou, J., Casalicchio, E., Spognardi, A. (eds.) ACNS 2020, Part I. LNCS, vol. 12146, pp. 208–227. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-57808-4_11
45. Jao, D., et al.: SIKE. Technical report, National Institute of Standards and Technology (2020)
46. Jao, D., De Feo, L.: Towards quantum-resistant cryptosystems from supersingular elliptic curve isogenies. In: Yang, B.-Y. (ed.) PQCrypto 2011. LNCS, vol. 7071, pp. 19–34. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-25405-5_2
47. Jiang, H., Zhang, Z., Chen, L., Wang, H., Ma, Z.: IND-CCA-Secure key encapsulation mechanism in the quantum random oracle model, revisited. In: Shacham, H., Boldyreva, A. (eds.) CRYPTO 2018, Part III. LNCS, vol. 10993, pp. 96–125. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-96878-0_4
48. Jiang, H., Zhang, Z., Ma, Z.: Key encapsulation mechanism with explicit rejection in the quantum random oracle model. In: Lin, D., Sako, K. (eds.) PKC 2019, Part II. LNCS, vol. 11443, pp. 618–645. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-17259-6_21
49. Kannwischer, M.J., Rijneveld, J., Schwabe, P., Stoffelen, K.: pqm4: post-quantum crypto library for the ARM Cortex-M4 (2021). https://github.com/mupq/pqm4
50. Kocher, P.C.: Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In: Koblitz, N. (ed.) CRYPTO 1996. LNCS, vol. 1109, pp. 104–113. Springer, Heidelberg (1996). https://doi.org/10.1007/3-540-68697-5_9
51. Kocher, P.C., Jaffe, J., Jun, B.: Differential power analysis. In: Wiener [78], pp. 388–397 (1999)
52. Kuchta, V., Sakzad, A., Stehlé, D., Steinfeld, R., Sun, S.-F.: Measure-rewind-measure: tighter quantum random oracle model proofs for one-way to hiding and CCA security. In: Canteaut, A., Ishai, Y. (eds.) EUROCRYPT 2020, Part III. LNCS, vol. 12107, pp. 703–728. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-45727-3_24
53. Lindner, R., Peikert, C.: Better key sizes (and attacks) for LWE-based encryption. In: Kiayias, A. (ed.) CT-RSA 2011. LNCS, vol. 6558, pp. 319–339. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-19074-2_21
54. Lyubashevsky, V., Peikert, C., Regev, O.: On ideal lattices and learning with errors over rings. In: Gilbert, H. (ed.) EUROCRYPT 2010. LNCS, vol. 6110, pp. 1–23. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-13190-5_1

55. McEliece, R.J.: A public-key cryptosystem based on algebraic coding theory. The deep space network progress report 42–44, Jet Propulsion Laboratory, California Institute of Technology, January/February 1978. https://ipnpr.jpl.nasa.gov/progress_report2/42-44/44N.PDF
56. Misoczki, R., Tillich, J., Sendrier, N., Barreto, P.S.L.M.: MDPC-McEliece: new McEliece variants from moderate density parity-check codes. In: ISIT 2013, pp. 2069–2073. IEEE (2013)
57. Naehrig, M., et al.: FrodoKEM. Technical report, National Institute of Standards and Technology (2020)
58. Niederreiter, H.: Knapsack-type cryptosystems and algebraic coding theory. Prob. Control Inf. Theory **15**(2), 159–166 (1986)
59. Okamoto, T., Pointcheval, D.: REACT: rapid enhanced-security asymmetric cryptosystem transform. In: Naccache, D. (ed.) CT-RSA 2001. LNCS, vol. 2020, pp. 159–174. Springer, Heidelberg (2000). https://doi.org/10.1007/3-540-45353-9_13
60. Pessl, P., Prokop, L.: Fault attacks on CCA-secure lattice KEMs. IACR TCHES **2021**(2), 37–60 (2021). https://tches.iacr.org/index.php/TCHES/article/view/8787
61. Qin, Y., Cheng, C., Ding, J.: An efficient key mismatch attack on the NIST second round candidate Kyber. Cryptology ePrint Archive, Report 2019/1343 (2019). https://eprint.iacr.org/2019/1343
62. Qin, Y., Cheng, C., Zhang, X., Pan, Y., Hu, L., Ding, J.: A systematic approach and analysis of key mismatch attacks on CPA-secure lattice-based NIST candidate KEMs. Cryptology ePrint Archive, Report 2021/123 (2021). https://eprint.iacr.org/2021/123
63. Rackoff, C., Simon, D.R.: Non-interactive zero-knowledge proof of knowledge and chosen ciphertext attack. In: Feigenbaum, J. (ed.) CRYPTO 1991. LNCS, vol. 576, pp. 433–444. Springer, Heidelberg (1992). https://doi.org/10.1007/3-540-46766-1_35
64. Ravi, P., Ezerman, M.F., Bhasin, S., Chattopadhyay, A., Roy, S.S.: Will you cross the threshold for me? - Generic side-channel assisted chosen-ciphertext attacks on NTRU-based KEMs. Cryptology ePrint Archive, Report 2021/718 (2021). https://eprint.iacr.org/2021/718
65. Ravi, P., Roy, S.S.: Side-channel analysis of lattice-based PQC candidates. NIST PQC Round 3 Seminars (2021). https://csrc.nist.gov/projects/post-quantum-cryptography/workshops-and-timeline/round-3-seminars
66. Ravi, P., Roy, S.S., Chattopadhyay, A., Bhasin, S.: Generic side-channel attacks on CCA-secure lattice-based PKE and KEMs. IACR TCHES **2020**(3), 307–335 (2020). https://tches.iacr.org/index.php/TCHES/article/view/8592
67. Rivest, R.L., Shamir, A., Adleman, L.M.: A method for obtaining digital signatures and public-key cryptosystems. Commun. Assoc. Comput. Mach. **21**(2), 120–126 (1978)
68. Saha, D., Mukhopadhyay, D., RoyChowdhury, D.: A diagonal fault attack on the advanced encryption standard. Cryptology ePrint Archive, Report 2009/581 (2009). https://eprint.iacr.org/2009/581
69. Saito, T., Xagawa, K., Yamakawa, T.: Tightly-secure key-encapsulation mechanism in the quantum random oracle model. In: Nielsen, J.B., Rijmen, V. (eds.) EUROCRYPT 2018, Part III. LNCS, vol. 10822, pp. 520–551. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-78372-7_17
70. Schwabe, P., et al.: CRYSTALS-KYBER. Technical report, National Institute of Standards and Technology (2020)

71. Shor, P.W.: Algorithms for quantum computation: discrete logarithms and factoring. In: 35th FOCS, pp. 124–134. IEEE Computer Society Press, November 1994
72. Shoup, V.: Using hash functions as a hedge against chosen ciphertext attack. In: Preneel, B. (ed.) EUROCRYPT 2000. LNCS, vol. 1807, pp. 275–288. Springer, Heidelberg (2000). https://doi.org/10.1007/3-540-45539-6_19
73. Singh, S.: The Code Book. Fourth Estate (1999)
74. Skorobogatov, S.P., Anderson, R.J.: Optical fault induction attacks. In: Kaliski, B.S., Koç, K., Paar, C. (eds.) CHES 2002. LNCS, vol. 2523, pp. 2–12. Springer, Heidelberg (2003). https://doi.org/10.1007/3-540-36400-5_2
75. Takahashi, A., Tibouchi, M.: Degenerate fault attacks on elliptic curve parameters in openssl. In: Euro S&P 2019, pp. 371–386. IEEE (2019)
76. Targhi, E.E., Unruh, D.: Post-quantum security of the Fujisaki-Okamoto and OAEP transforms. In: Hirt, M., Smith, A. (eds.) TCC 2016-B, Part II. LNCS, vol. 9986, pp. 192–216. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-53644-5_8
77. Vacek, J., Václavek, J.: Key mismatch attack on ThreeBears, Frodo and Round5. In: Hong, D. (ed.) ICISC 2020. LNCS, vol. 12593, pp. 182–198. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-68890-5_10
78. Wiener, M. (ed.): CRYPTO 1999. LNCS, vol. 1666. Springer, Heidelberg (1999). https://doi.org/10.1007/3-540-48405-1
79. Yen, S.M., Joye, M.: Checking before output may not be enough against fault-based cryptanalysis. IEEE Trans. Comput. **49**(9), 967–970 (2000)
80. Sung-Ming, Y., Kim, S., Lim, S., Moon, S.: A countermeasure against one physical cryptanalysis may benefit another attack. In: Kim, K. (ed.) ICISC 2001. LNCS, vol. 2288, pp. 414–427. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-45861-1_31
81. Zhang, X., Cheng, C., Qin, Y., Ding, R.: Small leaks sink a great ship: an evaluation of key reuse resilience of PQC third round finalist NTRU-HRSS. Cryptology ePrint Archive, Report 2021/168 (2021). https://eprint.iacr.org/2021/168. To appear in ICICS 2021