Don't Reject This: Key-Recovery Timing Attacks Due to Rejection-Sampling in HQC and BIKE

Qian Guo[3], Clemens Hlauschek[1,5], Thomas Johansson[3], Norman Lahr[2], Alexander Nilsson[3,4] and Robin Leander Schröder[1]

[1] Technische Universität Wien, {clemens.hlauschek,leander.schroeder}@inso.tuwien.ac.at

[2] Fraunhofer SIT, Darmstadt, Germany, norman@lahr.email

[3] Lund University, Lund, Sweden, {alexander.nilsson,qian.guo,thomas.johansson}@eit.lth.se

[4] Advenica AB, Malmö, Sweden

[5] RISE GmbH, Wien

Abstract. Well before large-scale quantum computers will be available, traditional cryptosystems must be transitioned to post-quantum (PQ) secure schemes. The NIST PQC competition aims to standardize suitable cryptographic schemes. Candidates are evaluated not only on their formal security strengths, but are also judged based on the security with regard to resistance against side-channel attacks. Although round 3 candidates have already been intensively vetted with regard to such attacks, one important attack vector has hitherto been missed: PQ schemes often rely on *rejection sampling* techniques to obtain pseudorandomness from a specific distribution. In this paper, we reveal that rejection sampling routines that are seeded with secret-dependent information and leak timing information result in practical key recovery attacks in the code-based key encapsulation mechanisms HQC and BIKE.

Both HQC and BIKE have been selected as alternate candidates in the third round of the NIST competition, which puts them on track for getting standardized separately to the finalists. They have already been specifically hardened with constant-time decoders to avoid side-channel attacks. However, in this paper, we show novel timing vulnerabilities in both schemes: (1) Our secret key recovery attack on HQC requires only approx. 866,000 idealized decapsulation timing oracle queries in the 128-bit security setting. It is structurally different from previously identified attacks on the scheme: Previously, exploitable side-channel leakages have been identified in the BCH decoder of a previously submitted HQC version, in the ciphertext check as well as in the pseudorandom function of the Fujisaki-Okamoto transformation. In contrast, our attack uses the fact that the rejection sampling routine invoked during the deterministic re-encryption of the decapsulation leaks secret-dependent timing information, which can be efficiently exploited to recover the secret key when HQC is instantiated with the (now constant-time) BCH decoder, as well as with the RMRS decoder of the current submission. (2) From the timing information of the constant weight word sampler in the BIKE decapsulation, we demonstrate how to distinguish whether the decoding step is successful or not, and how this distinguisher is then used in the framework of the GJS attack to derive the distance spectrum of the secret key, using 5.8×10^7 idealized timing oracle queries. We provide details and analyses of the fully implemented attacks, as well as a discussion on possible countermeasures and their limits.

Keywords: Timing Attack · Rejection Sampling · Fujisaki-Okamoto Transformation · Post-Quantum Cryptography · HQC · BIKE

1 Introduction

The progress in the research field of quantum computing weakens the previously estimated security guarantees of most currently deployed cryptographic primitives. In 2017, Michele Mosca [Mos17] estimated that the chance of having a large-scale quantum computer that breaks RSA-2048 to be 1/6 within a decade and 1/2 within 15 years; or even faster (6-12 years) by having massive investment, following Simon Benjamin [Ben17]. While such estimates and predictions are contested [Dya18, Kal20], it is important that the transition to post-quantum secure cryptographic algorithms happens well before an actual large-scale quantum computer is being built, as sensitive data might be stored for cryptanalysis at a later time, for example by surveillance infrastructure such the NSA's 3-12 exabyte data center in Utah [Hog15].

The security strengths of the new cryptographic primitives need to be evaluated with regard to possible attacks from classical as well as from quantum adversaries. But not only the algorithmic design need to withstand possible (theoretical) attacks, deployed schemes need to have secure implementations that withstand practical implementations attacks [HPA21], such as side-channel [Koc96, KJJ99, RKL+04] and fault attacks [BDL97, BDL01]. Not every cryptographic design has a straightforward elegant implementation that can be easily secured against all relevant implementation attacks. Daniel Bernstein and Tanja Lange repeatedly (e.g., in their analysis of the NIST ECC standards [BL16]) emphasize that a good cryptographic design requires simplicity of a secure implementation, and recommend that standardization bodies such as the National Institute of Standards and Technology (NIST) should require simplicity for secure implementations.

Timing attacks, first described by Kocher [Koc96], are arguably one of the most dangerous implementation attacks (right after more trivial, but still hard to spot, leakages such as the Heartbleed vulnerability [DKA+14]): An adversary just needs a communication channel to the target device and a precise timing measurement. It is often possible to mount an attack even remotely over the network [BB05, BT11, KPVV16, MSEH20, MBA+21], without physical access. Crosby et al. [CWR09] explore the limits of remote timing attacks. Often, timing leaks that have been mitigated against remote exploitation, such as the Lucky Thirteen attack [AP13] on TLS, can still be exploited in a Cloud/Cross-VM setup [AIES15]. These attacks exploit the timing variations which depend on the secret key material. When the timing variations include enough information the recovery of the secret key becomes possible.

In December 2016, the NIST announced a competition [oSN16] which aims to standardize schemes for post-quantum cryptography (PQC) and encouraged the authors to submit a reference implementation that addresses side-channel attacks in addition to the specification. NIST specifically motivates research to counter advanced side-channel attacks in the current, third round of the competition [MAA+20]. The two schemes Hamming Quasi-Cylic (HQC) [AAB+21] and the Bit Flipping Key Encapsulation (BIKE) [ABB+21] are promising code-based key encapsulation schemes and alternate candidates in the third round of the competition. As alternate candidates, they might be standardized by NIST in addition to the competition finalists in a fourth round. In its latest PQC standardization status report [MAA+20], NIST lauds HQC for its constant-time improvements, while voicing serious concerns over BIKE's side-channel protections and Indistinguishability under Chosen Ciphertext Attack (IND-CCA) security.

BIKE can be described as the McEliece scheme instantiated with Quasi-Cyclic Moderate Density Parity-Check (QC-MDPC) codes [MTSB13], using the equivalent Niederreiter scheme. The specification of BIKE is secure under the Indistinguishability under Chosen Plaintext Attack (IND-CPA) notion, where the security is related to some hard decoding problems in the Hamming metric. With an additional assumption on the probability of decryption errors that may occur in the BIKE decoding step, the scheme is shown to be IND-CCA secure, using the implicit-rejection variant of the Fujisaki-Okamoto (FO) transformation proposed by Hofheinz, Hövelmanns, and Kiltz (HHK) [HHK17]. Similarly, the Public Key Encryption (PKE) variant of HQC is secure under the IND-CPA notion. The Key Encapsulation Mechanism (KEM) variant of HQC utilizes another variant of the HHK FO transformation, converting the PKE variant to be secure with regard to IND-CCA. Many post-quantum secure schemes, e.g., code- and lattice-based, use the HHK transformation because it is resistant to the decryption errors that can occur in the decryption procedure of many non-traditional schemes. The attacks on BIKE as well as on HQC demonstrated in this paper are both possible due to the specific applications of the FO transformation to the respective underlying IND-CPA schemes. It is interesting to observe that the process of transforming an encryption scheme from a less secure to a more secure version introduces more complexity and hard-to-spot vectors for additional implementation vulnerabilities. However, our attacks require a static key setting and an active chosen-ciphertext attacker, a scenario where one would not employ just an IND-CPA secure scheme.

Related work. Our attacks, though structurally different from previous ones, build on a history of related side-channel attacks and cryptanalysis, leading to incrementally more secure and improved versions of the schemes. Recently, Wafo-Tapa et al. [WTBB⁺19] and Paiva et al. [PT19] present timing attacks on the non-constant time implementation of the Bose-Chaudhuri-Hocquenghem (BCH)-decoder of HQC. Both approaches exploit the dependence between the running time of the decoding procedure and the number of decoded errors. Paiva et al. require $400 \cdot 10^6$ decryption runs for the 128-bit security parameters. Wafo-Tapa et al. reach a key recovery after just 5441 calls with 93% success rate for the same security level. They proposed a constant-time BCH decoding to fix this issue.

Guo et al. [GJN20] show that the FO transformation of various proposed schemes is vulnerable to a timing attack by exploiting the comparison step in the decapsulation function, which is usually non-constant time (for example, when implemented via the `memcmp` function of the standard C library). The authors apply this timing attack to the lattice-based scheme FrodoKEM [NAB⁺20]. The attack requires 2^{30} decapsulation calls. They state that their attack is applicable to other proposed PQC schemes, among others, to HQC. They show the applicability to LAC [LLJ⁺19] in the appendix but do not explicitly show the effectiveness to HQC. The countermeasure to avoid the leakage is to use another constant-time comparison, e.g., as provided by OpenSSL[1]. However, in the same paper, Guo et al. describe how, more generally, any timing variance in the FO transformation that allows to distinguish between modifications that are below or above the error correction capability of the underlying primitive can in principle be used to mount key recovery attacks on IND-CCA secure KEM schemes. The research community seems to be acutely aware of the need to implement FO-based decapsulation methods in a constant time manner, as the source code of both HQC and BIKE, as well as recent discussions on the NIST PQC Forum[2] indicate.

An important attack for schemes based on QC-MDPC codes such as BIKE is the GJS attack [GJS16]. The attack uses an identified dependence between error patterns in decoding failures and the secret key. This attack assumes that the scheme is used in a static key setting requiring IND-CCA security. The *Error Amplification* attack [NJW18] builds on the GJS attack [GJS16], but improves it by using only a single initial error vector that results in a decoding failure and then modifies this in order to efficiently generate many more error vectors causing a decoding failure. These attacks are avoided in the BIKE scheme by selecting parameters such that the probability of a decoding failure for properly generated ciphertexts is very small.

[1] https://www.openssl.org/docs/man1.1.1/man3/CRYPTO_memcmp.html
[2] https://groups.google.com/a/list.nist.gov/g/pqc-forum/c/iVbJkCytoog

Most recently, Ueno et al. [UXT+22] explore a generic side-channel attack of the FO transformation commonly used in many PQC schemes: By exploiting side-channel leakage during non-protected Pseudorandom Function (PRF) execution in the re-encryption of the KEM decapsulation, they demonstrate that Kyber, Saber, FrodoKEM, NTRU, NTRU Prime, BIKE, SIKE, as well as HQC are vulnerable. The current reference implementation of HQC uses non-protected SHAKE as the relevant PRF.

Motivation. To the best of our knowledge, none of the previous attacks on HQC or BIKE, nor attack mitigations for those schemes, considered the non-constant time rejection sampling routine. Rejection sampling is one method to generate pseudorandom values from a specific distribution. The first side-channel attack targeting rejection sampling has been demonstrated at the CHES 2016 against the lattice-based signature BLISS [BHLY16], exploiting cache-access pattern from the Gaussian sampler. In contrast to the Gaussian samplers common in lattice-based schemes, HQC and BIKE use rejection sampling to generate random vectors with a specific Hamming weight. As we show in this paper, it turns out that the branching and thus the run-time of the rejection sampling routine in HQC and BIKE is indirectly in dependence relationship with the key.

This despite the fact that the current HQC specification states that the optimized reference implementation (using the vectorized Single Instruction Multiple Data (SIMD) instructions on an x86 machine) is now constant-time, and the source code is well analyzed concerning the leakage of any sensitive information. More specifically, the authors of HQC claim "to have thoroughly analyzed the code to check that only unused randomness (i.e. rejected based on public criteria) or otherwise nonsensitive data may be leaked." [AAB+21]. However, the specification reveals a subtle error: The modular design of the HQC KEM uses the FO transformation to transform an IND-CPA version of HQC into the IND-CCA KEM. This IND-CPA version is specified separately as a non-deterministic encryption scheme, where the Encrypt algorithm generates its randomness within the function. The specified KEM version then invokes a slightly different HQC.PKE encryption scheme that fixes the randomness via parameter passing to make the encryption deterministic. This subtle error in the specification might have hidden the fact that the rejection sampling invoked by the re-encryption step in the decapsulation routine has a dependence to the secret key: From reading this erroneous specification, it is easy to miss the fact that the rejection sampling in the Encrypt function is indeed dependent on the secret in the decapsulation. Adding further to the confusion, the plaintext message $\mathbf{m}$ can be chosen by the attacker in the IND-CPA scheme, and only becomes a secret-depending value in the IND-CCA KEM due the FO transformation.

On the other hand, the BIKE specification [ABB+21] demands (in the current version 4.2a, in Section 3.5 *Practical security considerations for using BIKE*) with regard to side-channel attacks only that the decoder must be implemented in constant time, but does not mention such considerations for the constant-weight hashing function, which is supposed to be implemented via rejection sampling.

While HQC uses non-constant time rejection sampling to generate pseudorandom vectors with a specific Hamming weight in the re-encryption step of the decapsulation due the employed variant of the HHK FO transformation, BIKE uses the implicit-rejection version of it, optimized so that it does not invoke the encryption function during the decapsulation. However, despite dispensing the re-encryption step, BIKE uses rejection sampling in the plaintext verification of the decryption step in the decapsulation routine.

Contributions. In this work, we analyze the current KEM variant of HQC and BIKE and show that they are still vulnerable to timing attacks. More specifically, we present

- hitherto unconsidered timing variations dependent on the secret key in the deterministic re-encryption of the KEM decapsulation of HQC, and in BIKE decapsulation in the plaintext-checking step of the decryption routine, both due to the non-constant time *rejection sampling* routines,

- a novel timing attack on the optimized reference implementation of HQC achieving a full secret key recovery with high probability,
- another novel timing attack on the existing implementations of BIKE achieving either a full secret key recovery or an efficient message recovery, where the key recovery attack uses the framework of the GJS attack to derive the distance spectrum of the secret key, and
- a discussion of possible countermeasures to avoid the identified leakage in the deterministic re-encryption step.

Timeline and Response. The attack on HQC, first described in the master thesis of one of our co-authors [Sch21] and then later in eprint 2021/1485/20211115:124514 [HLS21], and independently brought to the attention of NIST by the co-authors from Lund University has been acknowledged by the authors of HQC, who in response started working on more optimized countermeasures [Gab21]. We further identified that a similar side-channel exists in BIKE [Sch21, HLS21]. A concrete attack strategy exploiting the side-channel was discovered concurrently to our work and first published by BIKE co-author Nicolas Sendrier [Sen21]. To the best of our knowledge, we are the first to present a fully implemented version of an attack exploiting the identified side-channel. Also in parallel to the writing of this paper, Nicolas Sendrier [Sen21] presented a different approach than suggested here to mitigate the vulnerability, specific to BIKE: Sendrier first shows that the sampling routine we attack can be slightly biased in BIKE, which allows the use of a novel variant of Fisher-Yates sampling [Knu97, p.145] that is constant-time.

Organization. The remaining of the paper is described as follows. In Section 2, we give the preliminaries as well as a description of the specifications of the two KEM schemes HQC and BIKE, respectively. In Section 3, we explain the identified timing weakness in the functions that use rejection sampling and we describe in detail the full key recovery attack on HQC. In the same section we follow up by describing the details of another type of the attack applied on BIKE. In Section 4, we then present all the evaluation results from actually implementing the two previously described attacks. In Section 5, we discuss possible countermeasures and, finally, Section 6 concludes the paper.

2 Background

In this section, we introduce the background information on the schemes, HQC and BIKE, and the preliminaries that we require to explain our attacks in the following sections.

2.1 Preliminaries

We use a notation that we consider as close as possible to both the notations used by the HQC [AAB$^+$21] as well as the BIKE [ABB$^+$20] specification.

$\mathbb{F}_2$ denotes the binary finite field. Both HQC and BIKE use a cyclic polyomical ring, but with different parameters. So, for an integer $n \in \mathbb{Z}$ (resp., $r \in \mathbb{Z}$ in BIKE), we obtain the ring $\mathcal{R} = \mathbb{F}_2[X]/(X^n - 1)$, (resp. $\mathcal{R} = \mathbb{F}_2[X]/(X^r - 1)$). Elements in $\mathcal{R}$ will be represented by lower-case bold letters. These elements can be interchangeably considered as row vectors in a vector space over $\mathbb{F}_2$. Respective matrices will be represented by upper case bold letters. For $h \in \mathcal{R}$, let $|\mathbf{h}|$ denote the Hamming weight of a vector or polynomial $\mathbf{h}$.

Table 1: The HQC parameter sets [AAB+21]. The base Reed-Muller code is defined by $[128, 8, 64]$.

	RS-S			Duplicated RM						
Instance	n_1	k	d_{RS}	Mult.	n_2	d_{RM}	$n_1 n_2$	n	ω	$\omega_r = \omega_e$
hqc-128	46	16	31	3	384	192	17,664	17,669	66	75
hqc-192	56	24	33	5	640	320	35,840	35,851	100	114
hqc-256	90	32	49	5	640	320	57,600	57,637	131	149

2.2 Hamming Quasi Cyclic – HQC

HQC is a code-based post-quantum IND-CCA secure KEM. It is an alternate candidate in the third round of the NIST PQC competition [AAB+21]. Our work refers to the recent specification from June 2021. The HQC framework from which HQC stems was introduced by Aguilar et al. [AMBD+18]. Its security is reduced to problems related to the hardness of decoding random quasi-cyclic codes in the Hamming metric. The scheme uses a concatenated code $\mathcal{C}$ which combines an internal duplicated Reed-Muller code with the outer Reed-Solomon code. The resulting code has a publicly known generator matrix $\mathbf{G} \in \mathbb{F}_2^{k \times n_1 n_2}$.

The parameters are listed in Table 1 and we explain them in the following. The inner duplicated Reed-Muller code is defined by $[n_2, 8, n_2/2]$ and the outer, shortened Reed-Solomon code (RS-S) by $[n_1, k, n_1 - k + 1]$, with $k \in \{16, 24, 32\}$ depending on the corresponding security level. The concatenated code $\mathcal{C}$ is of length $n_1 n_2$. To avoid algebraic attacks the ambient space of vector elements is of length n which is the first primitive prime greater than $n_1 n_2$. It defines the polynomial quotient ring $\mathcal{R} = \mathbb{F}_2[X]/(X^n - 1)$.

HQC.PKE. The PKE variant of HQC consists of the Algorithms 1 to 3. The key generation in Algorithm 1 samples the elements $\mathbf{h}$, $\mathbf{x}$, and $\mathbf{y}$ from $\mathcal{R}$ uniformly at random where the Hamming weights of $\mathbf{x}$ and $\mathbf{y}$ are ω. The secret key sk consists of $\mathbf{x}$ and $\mathbf{y}$. The public key pk includes $\mathbf{h}$ and $\mathbf{s} = \mathbf{x} + \mathbf{h} \cdot \mathbf{y}$. The encryption function Algorithm 2 first samples the vectors $\mathbf{e}$ of weight ω_e as well as $\mathbf{r_1}$ and $\mathbf{r_2}$ of weight ω_r. The randomness of the sampling is seeded by the additional input θ. Therewith, the sampling becomes deterministic which is desired for the verification in the later decapsulation function. The ciphertext is a tuple with $\mathbf{u} = \mathbf{r_1} + \mathbf{h} \cdot \mathbf{r_2}$ and $\mathbf{v} = \mathbf{mG} + \mathbf{s} \cdot \mathbf{r_2} + \mathbf{e}$. The term $\mathbf{mG}$ in Line 6 corresponds to the encoding procedure of the concatenated code $\mathcal{C}$. It begins with the external Reed-Solomon code which encodes a message $\mathbf{m} \in \mathbb{F}_2^k$ into $\mathbf{m_1} \in \mathbb{F}_{2^8}^{n_1}$. Then the inner Reed-Muller code encodes each coordinate/byte $m_{1,i}$ into $\bar{\mathbf{m}}_{1,i} \in \mathbb{F}_2^{128}$ using $RM(1,7)$. Finally, $\bar{\mathbf{m}}_{1,i}$ is repeated 3 or 5 times depending on the security parameter to obtain $\tilde{\mathbf{m}}_{1,i} \in \mathbb{F}_2^{n_2}$. Thus, we get $\mathbf{mG} = \tilde{\mathbf{m}} = (\tilde{\mathbf{m}}_{1,0}, \ldots, \tilde{\mathbf{m}}_{1,n_1-1}) \in \mathbb{F}_2^{n_1 n_2}$. The decryption function in Algorithm 3 is to decode the term $\mathbf{v} - \mathbf{u} \cdot \mathbf{y}$ which results in

$$\begin{aligned} &(\mathbf{mG} + \mathbf{s} \cdot \mathbf{r_2} + \mathbf{e}) - (\mathbf{r_1} + \mathbf{h} \cdot \mathbf{r_2}) \cdot \mathbf{y} \\ =\ &\mathbf{mG} + (\mathbf{x} + \mathbf{h} \cdot \mathbf{y}) \cdot \mathbf{r_2} - (\mathbf{r_1} + \mathbf{h} \cdot \mathbf{r_2}) \cdot \mathbf{y} + \mathbf{e} \\ =\ &\mathbf{mG} + \mathbf{x} \cdot \mathbf{r_2} - \mathbf{r_1} \cdot \mathbf{y} + \mathbf{e}. \end{aligned}$$

Thus, the decoder has to correct the error

$$\mathbf{e}' = \mathbf{x} \cdot \mathbf{r_2} - \mathbf{r_1} \cdot \mathbf{y} + \mathbf{e}.$$

The decoding succeeds if $|\mathbf{e}'| \leq \delta$. The Decryption Failure Rate (DFR) denotes the probability when the weight exceeds the decoder's capacity.

```
Algorithm 1: HQC.KeyGen
    Input: param
    Output: sk, pk
1   $\mathbf{h} = \mathsf{Sample}(\mathcal{R})$
2   $\mathbf{x} = \mathsf{Sample}(\mathcal{R}, \omega)$
3   $\mathbf{y} = \mathsf{Sample}(\mathcal{R}, \omega)$
4   $\mathsf{sk} = (\mathbf{x}, \mathbf{y})$
5   $\mathsf{pk} = (\mathbf{h}, \mathbf{s} = \mathbf{x} + \mathbf{h} \cdot \mathbf{y})$
```

```
Algorithm 2: HQC.Encrypt
    Input: pk, $\mathbf{m}$, $\theta$
    Output: $c = (\mathbf{u}, \mathbf{v})$
1   $\mathsf{SampleInit}(\theta)$
2   $\mathbf{r_1} = \mathsf{Sample}(\mathcal{R}, \omega_r)$
3   $\mathbf{r_2} = \mathsf{Sample}(\mathcal{R}, \omega_r)$
4   $\mathbf{e} = \mathsf{Sample}(\mathcal{R}, \omega_e)$
5   $\mathbf{u} = \mathbf{r_1} + \mathbf{h} \cdot \mathbf{r_2}$
6   $\mathbf{v} = \mathbf{mG} + \mathbf{s} \cdot \mathbf{r_2} + \mathbf{e}$
```

```
Algorithm 3: HQC.Decrypt
    Input: $\mathsf{sk} = (\mathbf{x}, \mathbf{y})$
           $c = (\mathbf{u}, \mathbf{v})$
    Output: $\mathbf{m}$
1   $\mathbf{m} = \mathcal{C}.\,\mathsf{Decode}(\mathbf{v} - \mathbf{u} \cdot \mathbf{y})$
```

```
Algorithm 4: HQC.Encaps
    Input: pk
    Output: $K$, $(c, d)$
1   $\mathbf{m} = \mathsf{Sample}(\mathbb{F}_2^{n_1 n_2})$
2   $\theta = \mathsf{G}(\mathbf{m})$
3   $c = \mathsf{HQC.Encrypt}(\mathsf{pk}, \mathbf{m}; \theta)$
4   $K = \mathsf{K}(\mathbf{m}, c)$
5   $d = \mathsf{H}(\mathbf{m})$
```

```
Algorithm 5: HQC.Decaps
    Input: $\mathsf{sk} = (\mathbf{x}, \mathbf{y})$, $(c = (\mathbf{u}, \mathbf{v}), d)$
    Output: $K$
1   $\mathbf{m}' = \mathsf{HQC.Decrypt}(\mathsf{sk}, c)$
2   $\theta' = \mathsf{G}(\mathbf{m}')$
3   $c' = \mathsf{HQC.Encrypt}(\mathsf{pk}, \mathbf{m}'; \theta')$
4   if $c \neq c' \vee d \neq \mathsf{H}(\mathbf{m}')$ then
5       $K = \perp$
6   $K = \mathsf{K}(\mathbf{m}', c)$
```

HQC.KEM. The authors of HQC decided to use the Hofheinz-Hövelmanns-Kiltz (HHK) transformation [HHK17] to obtain an IND-CCA secure Key Encapsulation Mechanism from the IND-CPA secure PKE scheme described before. In contrast to the original FO transformation, the HHK approach is able to handle decryption failures. The KEM scheme may be used to share securely a random symmetric key K between two parties. The key generation is the same as for the PKE. The sender of a message applies the encapsulation function in Algorithm 4 to wrap a randomly chosen K and the receiver executes the decapsulation function in Algorithm 5 to obtain the same key or aborts if a decryption failure occurs.

The KEM construction requires the three independent cryptographic hash functions G, K, and H. To encapsulate a randomly chosen message $\mathbf{m}$ the randomness θ for the encryption is derived by $\mathsf{G}(\mathbf{m})$. The shared key K is a linkage of both the message $\mathbf{m}$ and the ciphertext c and is computed by $\mathsf{K}(\mathbf{m}, c)$. Finally, d is derived by computing the hash $\mathsf{H}(\mathbf{m})$.

In the decapsulation, the decryption function is invoked with the secret key sk and the ciphertext c to obtain the message $\mathbf{m}'$. To verify the ciphertext for integrity, a re-encryption of the message $\mathbf{m}'$ is performed using the randomness θ' derived from $\mathbf{m}'$. Then, the procedure checks whether the re-encrypted ciphertext c' matches the received c and whether the sent digest d equals the hash value of the decrypted message $\mathbf{m}'$. If this check succeeds, $\mathsf{K}(\mathbf{m}, c)$ is output, otherwise failure.

2.3 Bit Flipping Key Encapsulation – BIKE

BIKE is another code-based post-quantum KEM targeting IND-CCA security, which is also an alternate candidate in the third round of the NIST PQC competition. As other candidates, it has updated its specification from round to round and we consider the specification submitted to the most recent round of the NIST PQC competition (being round 3) [ABB+20]. It can briefly be considered as the McEliece scheme instantiated

Table 2: BIKE parameters.

Security	r	w	t	DFR
Level 1	12,323	142	134	2^{-128}
Level 3	24,659	206	199	2^{-192}
Level 5	40,973	274	264	2^{-256}

with QC-MDPC codes [MTSB13], using the equivalent Niederreiter scheme. Quasi-Cyclic Moderate Density Parity-Check codes are similar to more well known Quasi-Cyclic Low Density Parity-Check (QC-LDPC) codes, but parity checks have a somewhat larger weight ($O(n)$ instead of a constant, where n is the code length). The security of the scheme relies on quasi-cyclic variants of hard decoding problems from coding theory in the Hamming metric. The security level it provides is bounded by the complexity of solving these hard problems with the best known algorithms. In the case of BIKE we have a small probability of decryption failure, which gives on occurrence an error in decapsulation. Typically, a high security level demands that this probability of failure is negligible, say 2^{-128} or even smaller.

Let us now give a brief overview of the specification of BIKE. It is specified from three main values, being the Hamming weight of the error vector t, the row weight of the secret parity check matrix w, and the block length r. To achieve a given security level λ for IND-CPA security, the parameters t and w should be chosen according to the complexity of solving the underlying hard problems. To additionally achieve IND-CCA security, one need to make sure that the decryption failure probability is upper bounded by $2^{-\lambda}$. The block length r does not affect the computational hardness of the underlying problems much, but do affect the decryption failure probability.

In setup, one sets the target security level λ and the generates the parameters (r, t, w) and an additional parameter l, which gives the size of the shared key output, in bits. We also fix hash functions $\mathsf{H}, \mathsf{K}, \mathsf{L}$ and a decoder Decode. The message space is $\mathcal{M} = \{0,1\}^l$ and the shared key space is $\mathcal{K} = \{0,1\}^l$. We return to the hash functions later. BIKE uses the the cyclic polynomial ring $\mathcal{R} = \mathbb{F}_2/(X^r - 1)$; $\mathcal{H}_w$ is the secret key space $\mathcal{H}_w = \{(\mathbf{h_0}, \mathbf{h_1}) \in \mathcal{R}^2 : |\mathbf{h_0}| = |\mathbf{h_1}| = w/2\}$; finally $\mathcal{E}_t$ is the set of errors $\mathcal{E}_t = \{(\mathbf{e_0}, \mathbf{e_1}) \in \mathcal{R}^2 : |\mathbf{e_0}| + |\mathbf{e_1}| = t\}$.

The BIKE KEM consists of several algorithms. First, the key generation KeyGen is done as described in Algorithm 6. It creates the secret key sk, consisting of two low weight vectors $\mathbf{h_0}$ and $\mathbf{h_1}$ of length r, as well as a special value σ, used in case of error in decoding. It also creates the public key pk, being a length r vector computed as $\mathbf{h} = \mathbf{h_1}\mathbf{h_0}^{-1}$. The notation $\mathsf{Sample}(\mathcal{X})$ means that we uniformly pick an element from the set $\mathcal{X}$.

Next, the encapsulation algorithm Encaps outputs a ciphertext c that contains an encapsulated key value, using only the public key. It first selects a random bitstring m of length l. It then hashes this value to a weight t error vector $(\mathbf{e_0}, \mathbf{e_1}) \in \mathcal{E}_t$. Hashing is done using the special hash function H, which outputs weight t vectors. A ciphertext is then formed, where the first part is $\mathbf{c_0} = \mathbf{e_0} + \mathbf{e_1}\mathbf{h}$. The second part of the ciphertext is $c_1 = m \oplus \mathsf{L}(\mathbf{e_0}, \mathbf{e_1})$. We note that with knowledge of $(\mathbf{h_0}, \mathbf{h_1})$ in the secret key, one can efficiently reconstruct $(\mathbf{e_0}, \mathbf{e_1})$ from $\mathbf{c_0}$. Then one can also reconstruct m from c_1. The final step computes a shared key through $K = \mathsf{K}(m, c)$. All steps are illustrated in Algorithm 7.

The decapsulation algorithm Decaps is the final algorithm to describe. It outputs the shared key from the ciphertext c using the secret key. It first computes the error vector used to create $\mathbf{c_0}$ by $\mathbf{e}' = \mathsf{Decode}(\mathbf{c_0}\mathbf{h_0}, \mathbf{h_0}, \mathbf{h_1})$. Here Decode is a kind of bit-flipping decoder [Gal62]. The choice of decoder is a trade-off between efficiency and failure probability. In the BIKE specification, the Black-Gray-Flip (BGF) decoder is selected.

Algorithm 6: BIKE.KeyGen

```
Input: ·
Output: sk = $(\mathbf{h_0}, \mathbf{h_1}, \sigma)$
        pk = $\mathbf{h} \in \mathcal{R}$
1 $(\mathbf{h_0}, \mathbf{h_1}) = \mathsf{Sample}(\mathcal{H}_w)$
2 $\mathbf{h} = \mathbf{h_1}\mathbf{h_0}^{-1}$
3 $\sigma = \mathsf{Sample}(\mathcal{M})$
4 sk = $(\mathbf{h_0}, \mathbf{h_1}, \sigma)$
5 pk = $\mathbf{h}$
```

Algorithm 7: BIKE.Encaps

```
Input: pk = $\mathbf{h}$
Output: $K$, $c$
1 $m = \mathsf{Sample}(\mathcal{M})$
2 $(\mathbf{e_0}, \mathbf{e_1}) = \mathsf{H}(m)$
3 $c = (\mathbf{e_0} + \mathbf{e_1},$
      $m \oplus \mathsf{L}(\mathbf{e_0}, \mathbf{e_1}))$
4 $K = \mathsf{H}(m, c)$
```

Algorithm 8: BIKE.Decaps

```
Input: sk = $(\mathbf{h_0}, \mathbf{h_1}, \sigma)$
       $c = (\mathbf{c_0}, c_1)$
Output: $K$
1 $\mathbf{e}' =$
    $\mathsf{Decode}(\mathbf{c_0}\mathbf{h_0}, \mathbf{h_0}, \mathbf{h_1})$
2 $m' = c_1 \oplus \mathsf{L}(\mathbf{e}')$
3 if $\mathbf{e}' = \mathsf{H}(m')$ then
4 |   $K = \mathsf{K}(m', c)$
5 else
6 |   $K = \mathsf{K}(\sigma, c)$
7 end
```

Next $m' = c_1 \oplus \mathsf{L}(\mathbf{e}')$. If $\mathbf{e}'$ was correctly received, then $m' = m$. This is now checked by computing and comparing if $\mathbf{e}' = \mathsf{H}(m')$. If so, the shared key is set to $K = \mathsf{K}(m, c)$. The steps are illustrated in Algorithm 8.

3 Timing Attacks on HQC and BIKE

In this section we present the timing attacks on the schemes HQC and BIKE and the underlying vulnerabilities in both cases.

3.1 The Timing Attack on HQC

In the following, we show how the current optimized HQC implementation [AAB+] from June 2021 which is specified in [AAB+21] leaks timing information which enables the construction of a plaintext distinguisher. Then, this distinguisher is used as a plaintext-checking oracle within existing attacks described in [BDH+19] to achieve the key-recovery on the, now, deprecated version of HQC using BCH and repetition codes. Further, we propose an attack that enables the key-recovery on the current version using Reed-Solomon (RS) and Reed-Muller (RM) codes.

The vulnerability in the HQC implementations. As described in Section 2.2, the encryption function described in Algorithm 2 requires to sample bit vectors of a specified Hamming weight ω. The implementation of the sampling function uses rejection sampling to comply to the security properties, e.g., if a position is sampled twice. The runtime of the rejection sampling algorithm depends on the given seed θ. In the KEM version the en- and decapsulation procedures derive the seed for the encryption function from the message $\mathbf{m}$ by $\mathsf{G}(\mathbf{m})$. The dependence on the message in the decapsulation allows us to construct a plaintext distinguisher which we use to mount a timing attack afterwards.

The Sample function. The considered implementation of HQC implements the weighted vector sampling in the function `vect_set_random_fixed_weight`. For brevity we refer to this function as Sample. In each iteration the function generates random positions from the range $\{0, \ldots, n-1\}$ to set a bit at that position to 1 until w distinct bit positions have been sampled. Concretely, if the sampled bit position has already been sampled before the sample is rejected. Otherwise, the bit position is stored in an array. At the end, the vector of weight w is constructed by setting the bits at the w distinct positions that

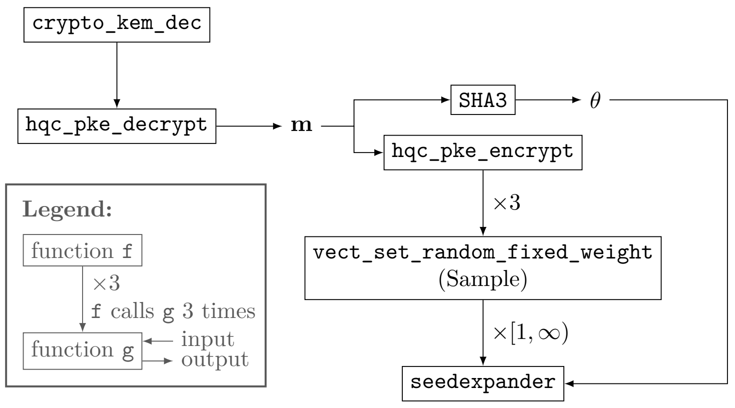


Figure 1: Visualization of the information flow in the decapsulation function of the current HQC KEM implementation [AAB+].

were sampled. The number of times a bit position collides with a previously sampled bit position is directly proportional to the runtime of the algorithm.

The randomness in the Sample function is deterministic and determined by an eXtendable-Output Function (XOF) implemented by the `seedexpander` function. For our analyses we assume that the outputs of the XOF are uniformly, independent and identically distributed (iid). The XOF influences the path that is taken through the function and is initialized with the seed $\theta = \mathsf{G}(\mathbf{m})$. The message $\mathbf{m}$ is obtained from the decoding of the ciphertext c, c.f., Line 1 in Algorithm 3. This data flow is illustrated in Fig. 1. Therefore, the message $\mathbf{m}$ controls how many iterations the rejection sampling algorithm takes. Further, a rejection leads to another call of the `seedexpander` function and, thus, to a large timing gap.

Additional `seedexpander` calls. We refer to `seedexpander` calls which are executed conditionally within the loop in the Sample function, c.f. Fig. 1, as *additional* `seedexpander` *calls*. For details, we refer to the original source code which can be found in the file `vector.c`, line 31, in [AAB+]. In general, unless otherwise specified, we only count the number of additional `seedexpander` calls and skip the default initial call. The `seedexpander` is initially used to produce $3 \cdot \omega_r$ bytes of randomness and store it into a buffer. If this randomness is sufficient to generate ω_r distinct bit positions, no additional `seedexpander` calls are issued. However, if even a single sample is rejected the algorithm will need to produce additional randomness by issuing another `seedexpander` call. The sampled bit positions are in the range of $\{0, \dots, n-1\}$. To generate these positions, the algorithm performs an inner rejection sampling algorithm. The inner rejection sampling algorithm samples a position from $\{0, \dots, 2^{24}-1\}$ that is to be reduced modulo n, where $n < 2^{24}$. However, the position is rejected if it is above the largest multiple of n that is smaller than 2^{24} which is defined by $\eta := \lfloor 2^{24}/n \rfloor n$ or `UTILS_REJECTION_THRESHOLD` in the implementation. This is to avoid biasing the distribution and discussed in detail in Section 5.2.

Thus, sampling distinct bit positions can fail in two ways: (1) The sampled position in $\{0, \dots, 2^{24}-1\}$ is larger than η or (2) it collides with a previously sampled one. We can model rejection sampling of a position as a Bernoulli variable with the success probability $p = \eta/2^{24}$. Each attempt to generate a valid bit position below n consumes 3 bytes of randomness. If the algorithm succeeds in picking a distinct bit position in every iteration, it does not need additional randomness. In this case `seedexpander` is not called within the for loop. However, if even a single sample fails or collides the algorithm will need to produce additional randomness, as it now requires more than $3 \cdot \omega_r$ bytes. The probability

Table 3: The approximated probabilities p for successfully sampling a bit position in the range required for unbiased modulo reduction, $\tilde{p}$ for completing the rejection sampling routine without exhausting the initially generated randomness, and for a message that causes at least 3 additional `seedexpander` invocations.

Instance	p (in %)	$\tilde{p}$ (in %)	$(1-\tilde{p})^3$ (in %)
hqc-128	99.94	81.95	0.58
hqc-192	99.79	65.93	3.95
hqc-256	99.97	79.09	0.91

of all ω_r samples succeeding and picking distinct positions out of n bit positions is

$$\tilde{p} = \prod_{i=0}^{\omega_r - 1} \left(p \frac{n-i}{n} \right)$$

which evaluates, for instance, to approx. 81.95% for the hqc-128 parameter set. Thus, only $1 - \tilde{p} \approx 18.05\%$ of all possible seeds θ result in at least one additional call to the `seedexpander` function. The probabilities for all parameter sets are listed in Table 3.

Decapsulation timing. Inspecting the decapsulation function in Algorithm 5 the timing variation is caused by the invocation of the encryption function using the seed $\theta = \mathsf{G}(\mathbf{m})$. Viewing the encryption function in Algorithm 2 we observe three calls to the previously discussed Sample function. One for each of the random vectors: $\mathbf{r_1}, \mathbf{r_2}, \mathbf{e}$, where the weight parameters ω_r and ω_e are equal. Each of these calls is using the same `seedexpander` instance, whose randomness depends upon the seed θ. In each of these three invocations there is a $1-\tilde{p}$ chance that `seedexpander` is called at least once within the for loop. Thus, $(1-\tilde{p})^3$ of messages result in three or more calls to `seedexpander`.

3.1.1 The Distinguisher

Given a ciphertext c we can distinguish whether the decrypted message $\mathbf{m}$ yields the same timing behavior during the encryption as another ciphertext. We define a distinguisher $\mathcal{D}$ as:

$$\mathcal{D}^{\mathcal{O}}(c_1, c_2) := \mathcal{O}(c_1) \stackrel{?}{=} \mathcal{O}(c_2) \tag{1}$$

where $\mathcal{O} = TB(\mathsf{sk}, \cdot)$ is the decapsulation timing oracle and yields the timing behavior – the number of `seedexpander` calls – of the provided ciphertext under the secret key sk and $\cdot \stackrel{?}{=} \cdot$ returns whether the two arguments are equal or not. The advantage of $\mathcal{D}$ when distinguishing a given ciphertext c_1 that decrypts to $\mathbf{m}_1$ from another ciphertext c_2 that decrypts to a uniform randomly chosen message $\mathbf{m}_2$ is given by:

$$\begin{aligned}
&|\Pr_{c_2 \xleftarrow{\$} \mathcal{C}}[\mathcal{D}^{TB(\mathsf{sk},\cdot)}(c_1, c_2) = 1 \mid \mathsf{Decrypt}(\mathsf{sk}, c_1) = \mathsf{Decrypt}(\mathsf{sk}, c_2)] - \\
&\quad \Pr_{c_2 \xleftarrow{\$} \mathcal{C}}[\mathcal{D}^{TB(\mathsf{sk},\cdot)}(c_1, c_2) = 1 \mid \mathsf{Decrypt}(\mathsf{sk}, c_1) \neq \mathsf{Decrypt}(\mathsf{sk}, c_2)]| \\
= &|\Pr_{c_2 \xleftarrow{\$} \mathcal{C}}[TB(\mathsf{sk}, c_1) = TB(\mathsf{sk}, c_2) \mid \mathsf{Decrypt}(\mathsf{sk}, c_1) = \mathsf{Decrypt}(\mathsf{sk}, c_2)] - \\
&\quad \Pr_{c_2 \xleftarrow{\$} \mathcal{C}}[TB(\mathsf{sk}, c_1) = TB(\mathsf{sk}, c_2) \mid \mathsf{Decrypt}(\mathsf{sk}, c_1) \neq \mathsf{Decrypt}(\mathsf{sk}, c_2)]| \\
= &\ 1 - \Pr_{c_2 \xleftarrow{\$} \mathcal{C}}[TB(\mathsf{sk}, c_1) = TB(\mathsf{sk}, c_2) \mid \mathsf{Decrypt}(\mathsf{sk}, c_1) \neq \mathsf{Decrypt}(\mathsf{sk}, c_2)]
\end{aligned}$$

where $\mathcal{C}$ is the ciphertext space. The last formula shows that the advantage is at a maximum when the probability of obtaining the same timing behavior for another ciphertext c_2 that decrypts to a different message is at a minimum. We can achieve this by minimizing the probability of the timing behavior of c_1 by picking a suitable message $\mathbf{m}_1$.

3.1.2 The Secret Key Recovery Attack

By using the observations of the vulnerability analysis to get a distinguisher described in Section 3.1.1 for a secret key recovery we propose the following attack idea. We pick a message $\mathbf{m}$ that has the property of resulting in 3 additional calls to the `seedexpander` function. Regarding the low probabilities in Table 3, we know that most of the messages do not share this property with our chosen message $\mathbf{m}$. Therefore, since we can determine whether a decryption has resulted in exactly 3 calls or not through the timing behavior, we can distinguish whether a ciphertext decrypts to the message $\mathbf{m}$ with high advantage. Next, we compute a ciphertext $c = (\mathbf{u}, \mathbf{v})$ by manually setting $\mathbf{r_1}$ to $1 \in \mathcal{R}$, and $\mathbf{r_2}$ and $\mathbf{e}$ to $0 \in \mathcal{R}$ during the encryption of $\mathbf{m}$. This ciphertext has the desirable property, that the error that the decoder has to correct during the decryption is just $\mathbf{y}$, a part of the secret key:

$$\mathbf{v} - \mathbf{u} \cdot \mathbf{y} = \mathbf{mG} + \mathbf{s} \cdot \mathbf{r_2} + \mathbf{e} - (\mathbf{r_1} + \mathbf{h} \cdot \mathbf{r_2}) \cdot \mathbf{y} = \mathbf{mG} - \mathbf{r_1} \cdot \mathbf{y} = \mathbf{mG} - \mathbf{y}. \qquad (2)$$

If we are able to find the error $-\mathbf{y} = \mathbf{y}$, we can compute the remaining part of the secret key as $\mathbf{x} = \mathbf{s} - \mathbf{h} \cdot \mathbf{y}$. Note, that we do not need $\mathbf{x}$ as it is never used during the decapsulation. Further, note that this ciphertext is not valid, since we cannot fully control $\mathbf{r_1}, \mathbf{r_2}$, or $\mathbf{e}$ during the encryption. For valid ciphertexts, these are derived from $\mathbf{m}$ via the XOF and the Sample function. We do not require a valid ciphertext, as our timing-side channel will reveal information, even if the ciphertext is rejected by the decapsulation oracle.

To recover the error $\mathbf{y}$ we follow the basic principles outlined by Hall et al. [HGS99]. The authors propose adding an error $\mathbf{e}'$ to the ciphertext c until we detect that the modified ciphertext c' decrypts to a different message $\mathbf{m}'$. Then, we test for every bit b in the ciphertext c', whether flipping it causes the ciphertext to decrypt back to the original message $\mathbf{m}$. If it does, we know that the bit b is an error bit in the modified ciphertext c'. Otherwise, b is not an error.

Unfortunately, we cannot directly apply this method to HQC for several reasons: (1) Instead of correcting errors we need to determine the error e of our original ciphertext $c = \mathbf{mG} + \mathbf{e}$. (2) Further, when flipping erroneous bits in the modified ciphertext it does not decrypt back to the original message in most cases. Thus, we would not detect that the bit is an error. (3) Finally, the timing side-channel can not distinguish pairs of messages that induce the same number of `seedexpander` calls. Therefore, we sometimes do not detect that our modified ciphertext does not decrypt to the same message $\mathbf{m}$ anymore.

The first issue can be solved by keeping track of the error $\mathbf{e}'$ that we add to c to obtain c'. If we flip a bit b in the ciphertext c' and it decrypts back to the original message $\mathbf{m}$, we know that b is an error in $c' = c + \mathbf{e} + \mathbf{e}'$. Let $\mathbf{e}'' = \mathbf{e} + \mathbf{e}'$. If the bit b is an error in $\mathbf{e}''$, then b is an error in $\mathbf{e}$ if and only if the b-th bit of $\mathbf{e}'$ is not set. Or in other words, if we did not introduce the error ourselves, we know that the bit is an error. Otherwise, we know that the bit is correct. The second issue vastly increases the number of timing oracle calls since it introduces a very high false negative rate. We do not gain any information if the ciphertext does not decrypt back to the original message. To address this issue, we retry the entire function multiple times, with many different $\mathbf{e}'$. Eventually, we obtain a decision for every bit. The third issue may be solved by obtaining three or more decisions for every bit, and then obtaining a final decision with a majority vote.

Our resulting attack approach is detailed in Algorithm 9. First, we need to find a proper message $\mathbf{m}$ which yields 3 additional `seedexpander` calls. Therefore, we perform an exhaustive but low effort search. According to Eq. (2), we apply the modified encryption to $\mathbf{m}$ to obtain the initial ciphertext $c = (\mathbf{u}, \mathbf{v})$. Further, we define a proper majority threshold T as the majority of N votes. Afterwards, we apply Algorithm 10 to find another

Algorithm 9: KeyRecovery

```
Input: Ciphertext $c$, parameter $N$ to compute majority threshold $T$.
Output: $\mathbf{y}$.
1  for $b = 0$ to $n_1 n_2 - 1$ do $\mathbf{y}[b] \leftarrow 0, \mathsf{t}[b] \leftarrow 0, \mathsf{r}[b] \leftarrow 0$
2  repeat
3      $(c', \mathsf{bs}) \leftarrow \mathsf{FindDiffMsg}(c)$
4      $\mathbf{e}' \leftarrow \mathsf{RecoverError}(c')$
5      majority $\leftarrow$ true
6      $T \leftarrow \lfloor \frac{N}{2} \rfloor + 1$
7      for $b = 0$ to $n_1 n_2 - 1$ do
8          if $\mathbf{e}'[b] = 1$ then
9              $\mathsf{t}[b] \leftarrow \mathsf{t}[b] + 1$
10             if $b \notin \mathsf{bs}$ then $\mathsf{r}[b] \leftarrow \mathsf{r}[b] + 1$
11         end
12         if $\mathsf{r}[b] < T$ and $\mathsf{t}[b] - \mathsf{r}[b] < T$ then
13             majority $\leftarrow$ false
14         end
15     end
16 until majority = true
17 for $b = 0$ to $n_1 n_2 - 1$ do $\mathbf{y}[b] \leftarrow \mathsf{r}[b] \geq T$
18 return $\mathbf{e}$
```

ciphertext $c' = (\mathbf{u}, \mathbf{y} + \mathbf{e}')$ and the corresponding $\mathbf{m}'$ that differs from $\mathbf{m}$. We only add $\mathbf{e}'$ to $\mathbf{v}$ because the input to the decoder evaluates to additional errors just in the secret key part $\mathbf{y}$, c.f., Eq. (3).

$$\text{Decrypt}(\text{sk}, (\mathbf{u}, \mathbf{v} + \mathbf{e}')) = \mathcal{C}.\,\mathsf{Decode}(\mathbf{v} + \mathbf{e}' - \mathbf{u} \cdot \mathbf{y}) = \mathcal{C}.\,\mathsf{Decode}(\mathbf{mG} + \mathbf{e}' - \mathbf{y}) \tag{3}$$

In particular c' should have exactly one more error bit than the decoder could correct. From this state, flipping any bit in c' and checking whether the ciphertext decodes again reveals whether that bit was an error bit in c or not. We can exploit this property to recover $\mathbf{y}$ later on. Starting from c and an error of $\mathbf{e}' = 0$, we iteratively increase the weight of $\mathbf{e}'$ by flipping single, random bits. After each flip, we send the modified ciphertext to the decapsulation timing oracle $\mathcal{D}^{\mathrm{TB}(\mathsf{sk},\cdot)}$ and check if the ciphertext causes a different amount of time in the decryption operation than our original ciphertext. If it does, we have found a ciphertext c' that decrypts to a different message $\mathbf{m}'$.

Then, for each bit position b in $\mathbf{v} + \mathbf{e}'$, we flip the bit and send $(\mathbf{u}, \mathbf{v} + \mathbf{e}' + 2^b)$ to the decapsulation timing oracle, where 2^b is a vector with the b^{th} bit set. If we detect that the timing is again equal to the timing of our original ciphertext, we assume that the decryption yields back the original message $\mathbf{m}$ and that the corresponding bit in the secret key part $\mathbf{y}$ is set. Otherwise, we assume that the ciphertext decrypts to a different message and that there is no error bit set at this position.

Finally, Algorithm 9 calls Algorithms 10 and 11 multiple times until a majority is revealed at each bit position for a 0- or 1-bit. To determine the majorities the counters in t record the total number of votes that have been cast for each bit b. The counters in r record the number of 1-votes for each bit b, i.e., the number of votes that the bit b is set. The number of 0-votes for a bit b is computed by $\mathsf{t}[b] - \mathsf{r}[b]$. For a majority either the number of 1-votes or the number of 0-votes has to exceed $\lfloor N/2 \rfloor + 1$.

Reducing the number of oracle queries. We can improve the attack by targeting a specific word of the duplicated RM code. Specifically, consider that the code used in HQC

Algorithm 10: FindDiffMsg

```
Input: $c$
Output: $c'$, flipped bits bs
1 $c' \leftarrow c$
2 bs $\leftarrow$ RandomPermutation($[0, \dots, n_1 n_2 - 1]$)

3 for $i = 0$ to $n_1 n_2 - 1$ do
4     Flip bit bs$[i]$ in $\mathbf{v}$ of $c'$
5     if $\mathcal{D}^{\mathrm{TB}(\mathsf{sk},\cdot)}(c, c') = 0$ then
6         return $(c',\ \mathsf{bs}[0, \dots, i])$
7     end
8 end
```

Algorithm 11: RecoverError

```
Input: Modified ciphertext $c'$
Output: Combined error $\mathbf{e}$
1 $\mathbf{e} \leftarrow \mathbf{0}$
2 for $i = 0$ to $n_1 n_2 - 1$ do
3     Flip bit $i$ in $\mathbf{v}$ of $c'$
4     if $\mathcal{D}^{\mathrm{TB}(\mathsf{sk},\cdot)}(c, c') = 1$ then
5         Set bit $i$ in $\mathbf{e}$
6     end
7     Flip bit $i$ in $\mathbf{v}$ of $c'$
8 end
```

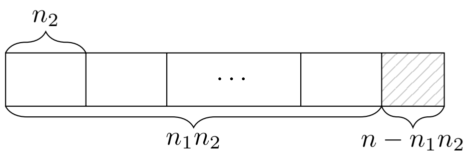


Figure 2: An element of $\mathbb{F}_2[x]/\langle x^n - 1\rangle$ and its segmentation into codewords of the inner code.

is a concatenated code combining an outer RS code with an inner duplicated RM code. During encoding, each element in the alphabet $\mathbb{F}_q$ from a word of the outer code is mapped to a message that the inner code can encode. We can obtain an oracle whether a word of the inner code decoded correctly by corrupting $\mathbf{v}$ such that a single additional corrupted inner code word would result in a decoding failure. We achieve this by corrupting δ – the error correction capacity of the outer code – elements of the outer code. We then may add an error $\mathbf{e}'$ to a single element of words of the RS code. A similar procedure has been previously described [BDH+19, Ex.15] to attack Lepton [YZ17] which uses BCH and repetition codes.

The oracle we construct here may also enable faster attacks [WTBB+19] if the noise learning problem [BDH+19] is solved for duplicated RM codes. We do not implement such a version of the attack as we are not aware of a solution to this problem.

Recovering the entire secret key. Using the methods described so far we can recover $n_1 n_2$ bits of the secret key $\mathbf{y}$. However, we are missing $n - n_1 n_2$ bits, that are required for using y during decryption. In Fig. 2 the structure of HQC codewords is displayed. Depending on the codes used, there are n_1 RM or repetition code codewords. However, $n - n_1 n_2$ bits of the n bits in total are never used during decoding. Thus, these bits cannot be obtained using the methods described so far. We now show how this situation can be remediated, and how it does not have a significant impact on the success probability, when the attack accounts for it. This issue was not addressed in some other attacks against HQC [WTBB+19]. Fortunately, the difference between n and $n_1 n_2$ is small for most parameters. However, for some parameters the difference could dominate the attack's complexity, if we were to brute force every possible combination. The largest difference with the new parameter sets is 37 bits in hqc-256. We can check whether a combination of bits is correct by checking whether we can decrypt an honestly encrypted message successfully. Fortunately, we can drastically reduce the search space while retaining a very high success probability. Assuming the number of bits set in the remaining bits is ≤ 2, the

Table 4: Remaining $n - n_1n_2$ bits that must be recovered for each parameter set, the number of ways to pick the remaining bits with a weight of up to 2, the probability that the weight is 0, and the probability that the weight is ≤ 2.

Instance	n_1n_2	n	ω	$n - n_1n_2$	$\sum_{i=0}^{2}\binom{n-n_1n_2}{i}$	$\Pr[Z = 0]$	$\Pr[Z \leq 2]$
hqc-128	17,664	17,669	66	5	16	$\approx$ 98.1 %	$\approx$ 100.0 %
hqc-192	35,840	35,851	100	11	67	$\approx$ 97.0 %	$\approx$ 100.0 %
hqc-256	57,600	57,637	131	37	704	$\approx$ 91.9 %	$\approx$ 100.0 %

number of ways to pick these bits is $\sum_{i=0}^{2}\binom{n-n_1n_2}{i}$. This number is low enough for all parameter choices to enumerate using a brute force search.

We now investigate the success probability given this dramatic search space reduction. We define $Y_{i,o,w}$ to be the number of elements that land inside a region of i elements when sampling w distinct elements uniformly from a region of $i + o$ elements. The region i (or "inside") corresponds to the bits that are set in the remaining $n - n_1n_2$ bits. The region o (or "outside") corresponds to the n_1n_2 bits that we have already obtained using the attack. Then the probability that x of the w distinct elements land inside the region of i elements is:

$$\Pr[Y_{i,o,w} = x] = \frac{\binom{i}{x}\binom{o}{w-x}}{\binom{o+i}{w}}$$

We now let $Z = Y_{n-n_1n_2,n_1n_2,\omega}$. Assuming the attack was successful for all n_1n_2 bits, the success probability is approx. 98.1% for hqc-128 when we guess that all remaining bits are zero, represented by the column $\Pr[Z = 0]$ in Table 4. However, this loss is preventable by brute-forcing the remaining bits. We can come very close to a success probability of 1, even for a modest search of only ≤ 2 set bits.

3.2 The Timing Attack on BIKE

Central elements for our attack are the decoding algorithm and the hash functions, which are described here a bit more. From the specification we see that the decoding step calls $\mathsf{Decode}(\mathbf{s}, \mathbf{h_0}, \mathbf{h_1})$ which returns either $(\mathbf{e_0}, \mathbf{e_1}) \in \mathcal{R}^2$ such that $\mathbf{e_0}\mathbf{h_0} + \mathbf{e_1}\mathbf{h_1} = \mathbf{s}$ or the failure symbol $\perp$. First, note that there is no restriction on the weight of the returned error in the Decode algorithm. Any weight is possible as long as $\mathbf{e_0}\mathbf{h_0} + \mathbf{e_1}\mathbf{h_1} = \mathbf{s}$. Secondly, if decoding is not successful and the failure symbol is returned, it has to be coded into a binary value. In existing reference implementations, failure is indicated by assigning a specific value like $(\mathbf{e_0}, \mathbf{e_1}) = 0$.

For the hash functions used, K and L are considered as standard hash functions, mapping to l-bit strings. But H is a special hash function, since its output is a vector of weight t. It finds its output by a rejection sampling method. Its description is given in Algorithm 12 and uses also a pseudorandom number generator called $\mathsf{AES\text{-}CTR\text{-}Stream}(\cdot)$[3] in the round-3 submission to NIST. In brief, the algorithm is producing a list of w different bit-positions in $\{0, ..., len - 1\}$, which correspond to the positions of the ones in the weight $t = w$ error vector of length $len = 2n$ that should be the output of the H hash function. The first step in the algorithm is to call $\mathsf{AES\text{-}CTR\text{-}Stream}(\cdot)$ to get a new position value and add it to the list if it was not previously already selected. The number of required calls for randomness (the final value of the i variable) varies, depending on the number of collisions with already selected values.

The situation in BIKE is very similar to the HQC case. Looking at the definition of $\mathsf{Encaps}/\mathsf{Decaps}$ (see Algorithms 7 and 8) we have seen that the rejection sampling takes

[3]In the most recent version, the designers instead employ SHAKE256.

Algorithm 12: WAES-CTR-PRF

```
Input: seed, $w$ (32 bits), len
Output: A list of $w$ different bit-positions in $\{0, ..., \text{len} - 1\}$.
1  wlist = $\phi$; ctr = 0; $i = 0$
2  $s$ = AES-CTR-Stream(seed, $\infty$)          // $\infty$ denotes "sufficiently large"
3  mask = $(2^{\lceil \log_2 r \rceil} - 1)$
4  while ctr < $w$ do
5      pos = $s[32(i+1) - 1 : 32i]$ & mask          // & denotes bitwise AND
6      if ((pos < len) AND (pos $\notin$ wlist)) then
7          wlist = wlist $\cup$ pos; ctr = ctr + 1;
8      end
9      $i = i + 1$
10 end
11 return wlist, $s$
```

place in the H function, in order to generate a random error vector of fixed-weight. A non-constant time implementation of H thus means that we can distinguish between the cases $m = m'$ and $m \neq m'$ with some probability. The value m' directly depends on the ability of the decoder to correctly extract the error vector $e = (\mathbf{e_0}, \mathbf{e_1})$ from the ciphertext c. This means that we have, as for the case of HQC, a distinguisher between chosen ciphertexts above and below the error correction capability of the decoder. This assumes the rejection sampling algorithm is not implemented in constant time, of course. BIKE officially claims only a IND-CPA secure scheme with the ephemeral key use-case, although they claim IND-CCA security if the decoder they use can be shown to have a decoding failure rate lower than the bit-security level of the scheme.

We are now ready to formulate an attack on BIKE, based on the described observations. As before, we consider an IND-CCA scenario, where we assume that we can compute ciphertexts (with encapsulated keys), feed a ciphertext for decapsulation and observe the output of decapsulation. This may for example be an attempt to establish a joint key. As this is a timing attack, we also add the assumption that we get timing information from the decapsulation step.

We leverage the GJS attack [GJS16, NJW18] and use the rejection sampling vulnerability as way to act as a distinguisher of decoding failure. This attack assumes that the scheme is used in a static key setting requiring IND-CCA security. The *Error Amplification* attack [NJW18] builds on the GJS attack [GJS16], but requires only a single initial error vector that results in a decoding failure and then modifies this in order to generate many more error vectors. Let us give very brief descriptions of these attacks.

3.2.1 The GJS Attack

The GJS attack [GJS16, NJW18] was described as an attack on QC-MDPC public-key schemes, using decryption failures that occur. As BIKE is a QC-MDPC scheme, the attacks are directly applicable. In our case, the secret key is $(\mathbf{h_0}, \mathbf{h_1})$, which also determines the secret parity-check matrix of the code to be decoded. Central is the notion of distance between two ones at position i_1 and i_2, $i_1 < i_2$, in a vector. It is defined as the smallest value of $(i_2 - i_1)$ and $(i_1 - i_2) + r$, where r is the length of the vector (the smallest distance between the two ones in cyclic sense).

The *distance spectrum* for a length r vector x is denoted $D(x)$. It is (in its simplest form) defined as

$$D(x) = \{d : 1 \leq d \leq r/2, d \text{ is a distance existing in } x\}.$$

Algorithm 13: Key recovery from distance spectrum (from [GJS16])

```
Input: distance spectrum $D(\mathbf{h_0})$, partial secret key $\mathbf{h_0}$, current depth $l$
Output: recovered secret key $\mathbf{h_0}$ or message "No such secret key exists"
1  Initial recursion parameters: distance spectrum $D(\mathbf{h_0})$, empty set for secret
   key, current depth 0
2  if $l = w$ then
3      return $\mathbf{h_0}$                                        // secret key found
4  end
5  for all potential key bits $i$ do
6      if all distances to key bit $i$ exist in $D(\mathbf{h_0})$ then
7          Add key bit $i$ to secret key $\mathbf{h_0}$
8          Make recursive call with parameters $D(\mathbf{h_0}), \mathbf{h_0}$ and $l + 1$
9          if recursive call finds solution $\mathbf{h_0}$ then
10             if $\mathbf{h_0}$ is the secret key then
11                 return $\mathbf{h_0}$                            // secret key found
12             end
13         end
14         Remove key bit $i$ from secret key $\mathbf{h_0}$
15     end
16 end
17 return "No such secret key exists"
```

It can be extended by also introducing $\mu(d)$, where $\mu(d)$ is the number of times the distance d is present in vector x, when $d \in D(x)$.

The approach is now to examine the decoding result for different error patterns. In particular, one picks errors from special subsets. For example, let Ψ_d be the set of all binary vectors of length $n = 2r$ having exactly t ones, where all ones are placed with distance d in the first half of the vector. The other half of the vector is zero. The construction of Ψ_d gives repeated ones at distance d at least $t/2$ times, where

$$\begin{aligned}\Psi_d = \quad & \{(\mathbf{e}, \mathbf{0}) \mid \exists \text{ distinct } s_1, s_2, \ldots, s_t, \text{ s.t. } \mathbf{e}_{s_i} = 1, \text{ and} \\ & s_{2i} = (s_{2i-1} + d) \bmod r \text{ for } i = 1, \ldots, t/2, \text{ and } |\mathbf{e}| = t\}\end{aligned} \tag{4}$$

In the attack phase one sends many messages with the error selected from the subset Ψ_d. When there is a decoding error one records this. With enough samples one can compute an empirical decoding error probability for the subset Ψ_d. Furthermore, this is done for $d = 1, 2, \ldots, r/2$. The main observation is that there is a strong correlation between the decoding error probability for error vectors from Ψ_d and the existence of a distance d between two ones in the secret vector $\mathbf{h_0}$. If there exists two ones in $\mathbf{h_0}$ at distance d, the decoding error probability is much smaller than if distance d does not exist between two ones.

After sending many messages, we look at the decoding error probability for each Ψ_d and classify each d, $d = 1, 2, \ldots, U$ according to its multiplicity $\mu(d)$, since each distance can appear not only once but many times. This provides a distance spectrum for the secret vector $\mathbf{h_0}$, which we write $D(\mathbf{h_0})$. Finally, from $D(\mathbf{h_0})$ it is an easy task to compute $\mathbf{h_0}$. One can even have a smaller number of wrong values in $D(\mathbf{h_0})$ and still be able to compute $\mathbf{h_0}$. We list the basic key reconstruction algorithm from [GJS16] in Algorithm 13 for completeness. The advanced version that is capable of recovering keys from distance spectrum with errors is proposed in [GJW19].

Algorithm 14: Pseudo code of attack: BikeAttack(h, w^*, M, I)

```
                                                            // Preamble
1  $m \leftarrow$ Plaintext such that $\mathsf{H}(m)$ is easily distinguishable by timing attack
2  while True do
3      Generate random $\mathbf{e}$, of hamming weight $w^*$
4      if DecodingFailureDistinguisher($\mathbf{e}, I$) = True then
5          break                  // Found the first $\mathbf{e}$ which causes a decoding failure
6      end
7  end
                                                            // Main body
8  $F, G, A, B \leftarrow$ empty vectors
9  for $i \leftarrow 1, M$ do
10     $\mathbf{e}^* \leftarrow$ Move random non-zero bit in $\mathbf{e}$
11     $\Delta D_d \leftarrow$ Distance spectrum differences between $\mathbf{e}$ and $\mathbf{e}^*$
12     if DecodingFailureDistinguisher($\mathbf{e}^*, I$) = True then
13         $\mathbf{e} \leftarrow \mathbf{e}^*$
14         Update lists $F, G$ with $\Delta D_d$ according to [NJW18]
15     else
16         Update lists $A, B$ with $\Delta D_d$ according to [NJW18]
17     end
18 end
                                                            // Postamble
19 $A' \leftarrow \max(A) - A + \min(A)$
20 $D \leftarrow F + G + A' + B$
21 Recover secret key with distance spectrum $D$ as per [GJS16]
```

3.2.2 The Secret Key Recovery Attack

The attack now follows the procedure listed in Algorithm 14 and using the distinghuisher in Algorithm 15. Let us step through the different parts of the attack in more detail.

1. We start by finding m such that $\mathsf{H}(m)$ is easily distinguishable by a timing attack. Here we pick m with an extraordinarily distinct timing profile (i.e. long or short) in the *rejection sampling.* It means that we have many or few collisions in Algorithm 12 that makes execution require more or less time, than the average case. Selection of strategy, as well as details on the number of calls, will be discussed later.
2. Construct an error pattern $e = (\mathbf{e_0}, \mathbf{e_1})$ with higher than normal Hamming weight so that we are as close to the decoding limit as possible. We assume one part of e (w.l.o.g., $\mathbf{e_1}$) is an all-zero vector.
3. Calculate c and transmit to target.
4. Determine if $m' \overset{?}{=} m$ by timing attack.
5. Repeat from step 2. to collect many e where $m' \neq m$ as per GJS attack or *Error Amplification* attack
6. After sufficiently many errors are collected, we could determine the distance spectrum statistically.
7. The secret key can be recovered via the reconstruction method in [GJS16] or the improved reconstruction method in its extended version [GJW19] that can handle errors in the recovered distance spectrum.

Algorithm 15: DecodingFailureDistinguisher($\mathbf{e}, I$)

```
1  $\mathbf{e_0}, \mathbf{e_1} \leftarrow \mathbf{e}$
2  $c \leftarrow (\mathbf{e_0} + \mathbf{e_1}\mathbf{h}, m \oplus \mathsf{L}(\mathbf{e_0}, \mathbf{e_1}))$
3  $S \leftarrow \emptyset$
4  for $i \leftarrow 1, I$ do
5      start $\leftarrow$ RDTSCP
6      BIKE.Decaps(c)
7      stop $\leftarrow$ RDTSCP
8      $S \leftarrow S \cup$ (stop $-$ start)
9  end
10 Determine decoding failure $f$ or not with $S$
11 return $f$
```

We can check the correctness of this approach. First, L takes input from $\mathcal{R}^2$, so it delivers a result for any choice of $e = (\mathbf{e_0}, \mathbf{e_1})$. Hence we can build ciphertexts accordingly. In the decaps step, $\mathbf{c_0} \in \mathcal{R}$ is always a valid input to the decoder. The decoder delivers an error $\mathbf{e}'$ or a failure. But since the result from the decoder is fed into the L with input in $\mathcal{R}^2$, the failure symbol must be interpreted as a fixed value in $\mathcal{R}^2$ (as is also done in the reference code). Altogether, there are no problems with the domain and range of functions. We can feed decaps with ciphertext corresponding to error vectors with higher weight than specified. It is only in the last check of $\mathbf{e}' = \mathsf{H}(m')$ that it will fail, since H is only delivering error vectors of weight t.

4 Evaluation

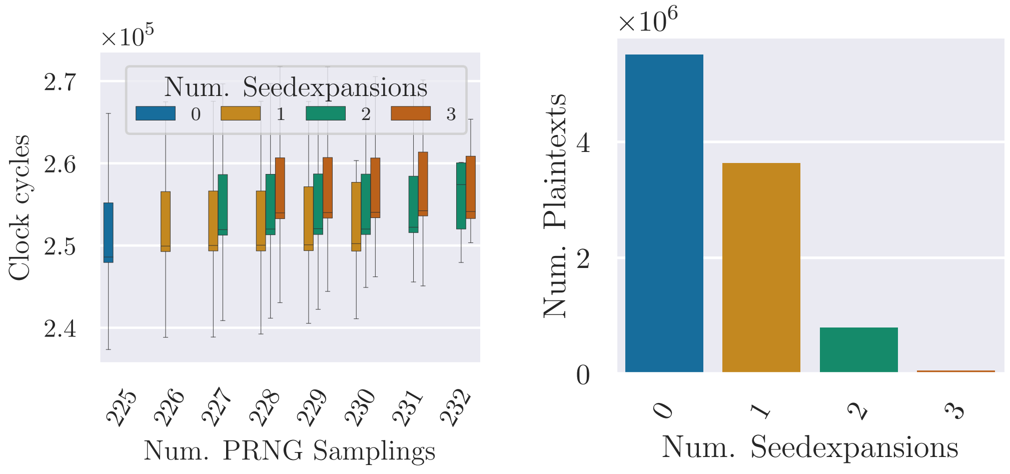
In this section we present the empirical evaluations of both attack approaches described in Sections 3.1 and 3.2.

4.1 Empirical Evaluation of the Attack on HQC

To confirm the previously postulated hypothesis about the timing behavior we performed a leakage assessment by measuring the CPU cycles of the decapsulation function in the hqc-128 setting for ten million random ciphertexts. Fig. 3a shows how more `seedexpander` calls result in an increased running time. We observe up to 3 additional `seedexpander` calls. In Fig. 3b we can see the frequency of different timing behaviors. As expected, the frequency decreases when the number of additional `seedexpander` calls increases. Further, the rate of three additional calls is low enough to be distinguishable to the other three cases. The probability of four additional calls is negligible and does not occur.

We have empirically verified the existence of the timing variation by generating random ciphertexts under a single keypair and measuring the number of cycles that the decapsulation algorithm required for 100 random ciphertexts. To measure the number of cycles that an operation takes we use the `rdtsc` instruction on x86 as recommended by Intel [Pao10]. Section 5.5 shows whether there is a difference in decapsulation time between pairs of 100 ciphertexts generated for a single keypair. We determine whether there is a statistically significant difference using Welch's t-test [Wel47] ($\alpha = 0.1\%$). The t-statistic for two distributions X_1 and X_2 in Welch's t-test is computed as:

$$\frac{\bar{X}_1 - \bar{X}_2}{\sqrt{\frac{s^2_{X_1}}{N_1} + \frac{s^2_{X_2}}{N_2}}} \tag{5}$$



(a) For each observed combination of the number of additional `seedexpander` calls and the number of times a position was attempted to be sampled we show a boxplot of the number of cycles that the decapsulation function took.

(b) Bar plot of the number of additional `seedexpander` calls observed for the 10 million random ciphertexts generated. 3 additional `seedexpander` calls corresponds to the rarest observed timing behavior.

Figure 3: Decapsulation timings and frequency of different timing behaviors. We observe that the running time of the decapsulation function is proportional to the number of seedexpansions and that more `seedexpander` calls are rare. The left figure shows a standard box plot with the median indicated within the the box, which also shows the quartiles. The whiskers extend to show 1.5 times the interquartile range.

where $\bar{X}_i$, $s^2_{X_i}$ and N_i are the sample mean, variance and size of X_i, respectively. The degrees of freedom are estimated by the Welch-Satterthwaite equation:

$$\nu = \frac{\left(\frac{s^2_{X_1}}{N_1} + \frac{s^2_{X_2}}{N_2}\right)^2}{\frac{\left(\frac{s^2_{X_1}}{N_1}\right)^2}{N_1-1} + \frac{\left(\frac{s^2_{X_2}}{N_2}\right)^2}{N_2-1}}. \tag{6}$$

The results show that many pairs of ciphertexts emit a statistically significant difference in decapsulation time. We have performed the same test again focussing only on the `seedexpander` function and achieve very similar results.

We implemented the optimized attack against hqc-128 using an idealized timing oracle that reveals the number of `seedexpander` calls during the decapsulation. The attack may be implemented analogously for the other parameter sets. We set $N = 5$ for the number of samples from which a majority must be formed for each bit. We performed the attack 6096 times in 114 CPU core hours on a Ryzen 5900X with 64 GiB DDR4 3600 MT/s CL18 RAM. Each attack required a median of 866,143 idealized timing oracle calls. Of the 6096 attacks 5315 were successful, yielding a success rate of more than 87 %. Among the failed attacks, approx. 26 % terminated with less than 3 incorrect bits in the secret key component $\mathbf{y}$. An additional brute-force step comprised of approx. $\sum_{i=0}^{3} \binom{17{,}669}{i} \approx 2^{40}$ offline decapsulations could therefore further boost the success probability. Furthermore, approx. 86 % of the failed attacks terminated with less than 20 incorrect bits and could therefore drastically reduce the security level of HQC. Thus, even if we are not able to recover all bits of the secret key we deem it likely that one can apply the known attacks

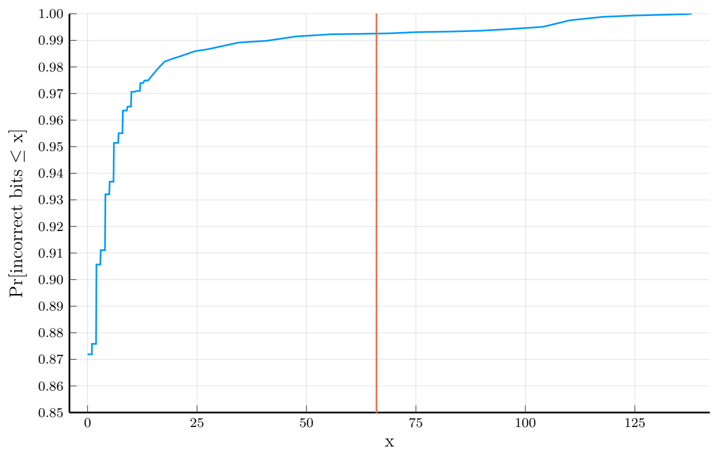


Figure 4: Empirical cumulative distribution function of the number of incorrect bits during the attacks. Approx. 87 % succeeded immediately. For those that failed additional post-processing steps could further improve the success probability. The vertical line indicates the weight of the secret key $\mathbf{y}$. Less than 1 % of cases the attack terminated with more incorrect bits than bits are set in the secret key.

to the HQC scheme which are listed in [AAB+21] as it will become feasible to solve the syndrome decoding problem or to mount structural attacks. We empirically determined the probability distribution of the number of incorrect bits after an attack and show the cumulative distribution function in Fig. 4.

4.2 Empirical Evaluation of the Attack on BIKE

The BIKE specification changed in some major ways between round 2 and 3 and there are now only a single bike variant with different security levels. The submitted version of the specification is 4.1. The specification has been updated during round 3 to version 4.2 in ways relevant to our attack; specifically the PRNG function used by the H function has been replaced. Though, the side-channel and the presented attack remains, so the changes are not detailed in this paper.

The presented version of the BIKE attack assumes the following pre-conditions:

1. It is possible to generate a decoding failure (and then use the *Error Amplification* attack to generate a chain of related decoding failures). This is possible due to
    - increasing the error weight when crafting the modified ciphertext artificially increases the DFR.
    - the lack of mandated weight-check on the error vector in the decapsulation stage of the BIKE specification.[4]
2. The timing profile of the H function depends on its input (value of m), i.e. it is not (or insufficently) protected against side-channels.
3. The attack requires a IND-CCA setting with static key re-use.

We now list some existing implementations and discuss the applicability of our attack:

[4]a weight check is discussed in the *Design Rational* chapter, but left out in the *Specification* chapter. It is mentionend in the IND-CCA security reduction to be implicit in the $\mathbf{e}' = \mathsf{H}(m')$ check, but not in the specification of H.

- **Reference implementation**: All versions of the reference implementation of BIKE fulfills these pre-conditions. But since it is not designed to protect against side-channels the existence of an attack is not unexpected.

- **Protected additional implementation**: It should be noted that there is no submitted official *additional implementation* in the submission package[5] for NIST PQC Round 3 version of the BIKE specification. The *additional implementation* folder in the Round 3 submission package is the protected implementation of the BIKE round 2 specification, version 3.2[6]. This version is vulnerable to the attack presented in this paper, with some minor modifications.

- **Github version**: Located at github.com/awslabs/bike-kem/ is another implementation, which has an additional weight check on the error vector (not given in the BIKE specification), located before the call to the H function. The end result is that in case of decoding failure *or* an error pattern of weight $\neq t$, the input to H is randomized. This renders the described attack much more difficult to exploit, since we are required to find a decoding failure without changing the weight of the error vector. On the other hand, the extra weight check opens up an even more efficient message recovery attack and we provide a very brief description of this in Section 4.3.

  Liboqs from the Open Quantum Safe Project [SM16] appears to use the same version of the BIKE implementation as the one above. Also, a recent **3rd party Intel Haswell** implementation due to [CCK21] that targets the Intel Haswell family of CPUs to achieve greater speeds than the official implementation appears to be based on the Github version and have copied the additional weight check.

- **3rd party ARM Cortex M4**: In the same paper [CCK21], the authors present a side-channel protected implementation targeting the ARM Cortex M4 processor. This version does not employ the additional weight check and is thus vulnerable to our attack.

The simulations and experiments related to BIKE in this paper is using the liboqs implementation for BIKE version 4.1, with the additional weight checks turned off. This enables us to verify our attack in a close to real world scenario. Ideally, the experiments would also be performed on the unmodified 3rd party ARM Cortex M4 implementation. Due to time-constraints however, we restricted ourselves to the Intel x86 platform-based implementations of the submitted BIKE version.

The empirical investigation into *BIKE-L1* shows the number of expected rejections by the rejection sampling algorithm in Fig. 5. From the experiment we draw the following conclusions about the targeted implementation:

1. The number of PRNG samplings θ are equal to the number of sampled bit positions in $\mathbf{e_0}, \mathbf{e_1}$, therefore $\theta \geq T$.

2. θ, for *BIKE-L1*, has an expected value $E(\theta) \approx 178.6$, over the space of $\mathcal{M}$.

3. The number of rejections for *BIKE-L1* has an expected value of $E(\theta - T) = 44.6$

4. The experiment shows a skewed[7] normal distribution with a standard deviation of $\sigma = 7.714$

[5]obtained on 2021-12-31 from https://csrc.nist.gov/CSRC/media/Projects/post-quantum-cryptography/documents/round-3/submissions/BIKE-Round3.zip

[6]the same is true for the additional implementation found on the bikesuite.org website (Last checked 2021-12-31)

[7]Skewed due to the influence of condition $\theta \geq T$

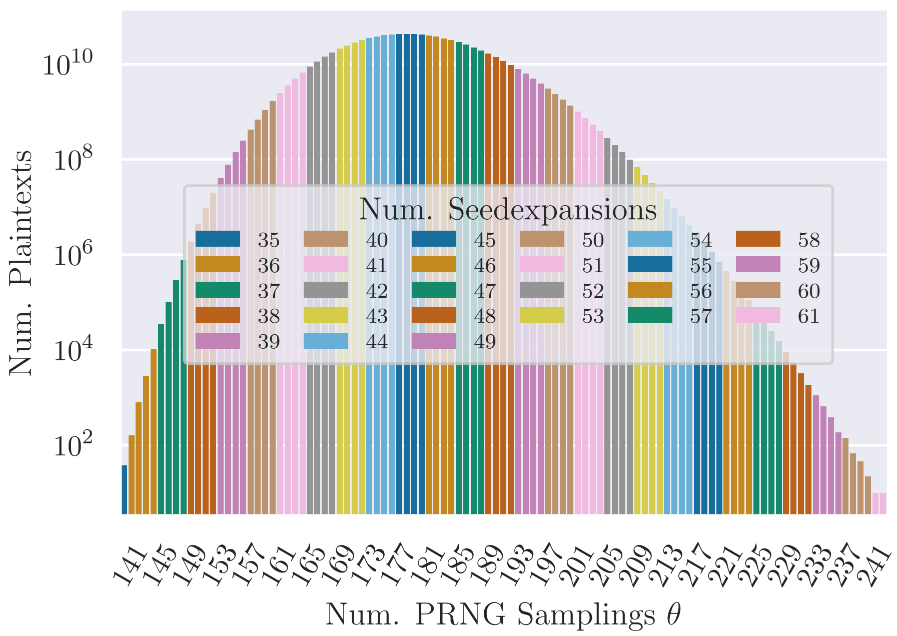


Figure 5: *BIKE-L1*: Distribution of the number of samplings θ in H to the underlying PRNG function as empirically simulated with approximately 10^{11} randomly generated plaintexts. Also presented are the number of seed expansions to the PRNG function.

5. The underlying PRNG can serve 4 random samplings before requiring a new seed expansion.

To determine the existence of the timing side-channel we observe the timing distributions of the decapsulation method (Algorithm 8) as a function of θ. As we can see in Fig. 6 it is indeed possible to distinguish between a high and low θ, although the variations are slight and we therefore require a relatively large number measurements for the distinguisher to give accurate outputs.

We performed the BIKE experiments in an HP EliteBook 820-G4 notebook with Intel Core i5-7200@2.50GHz and 8Gb RAM running on Ubuntu 20.04 LTS. We set Linux scaling governor to 'performance', turned off hyper threading, and turned off all extraneous processes.

Interestingly, the number of seed expansion calls does not appear to provide any noticeable influence on the runtime of the implemented rejection sampling algorithm. Consequently we must select a plaintext based on θ alone. Each seed expansion call is a simple call into AES which generally is implemented using the Advanced Encryption Standard New Instructions (AES-NI) CPU instruction set extension, which is very fast on modern CPUs.

As previously noted, since it is a one-time pre-computational cost, we can spend an almost arbitrary amount of computation looking for a good candidate plaintext which will provide us with a distinct timing profile in the rejection sampling algorithm.

Since we do not control what new plaintext is generated by decoding failures our candidate plaintext must cause a timing profile which is measurably distinct from other *likely* plaintexts. Likely, in this context, relates to the notion of the probability of making erroneous decisions for $m' \overset{?}{=} m$. That is, if $m' \neq m$ and $|\theta_m - \theta_{m'}| < \Delta\theta$, for some value $\Delta\theta$ for which the distinguisher is no longer reliable, then the distinguisher may output the wrong decision. The probability of the decoder randomly returning such an m' is the

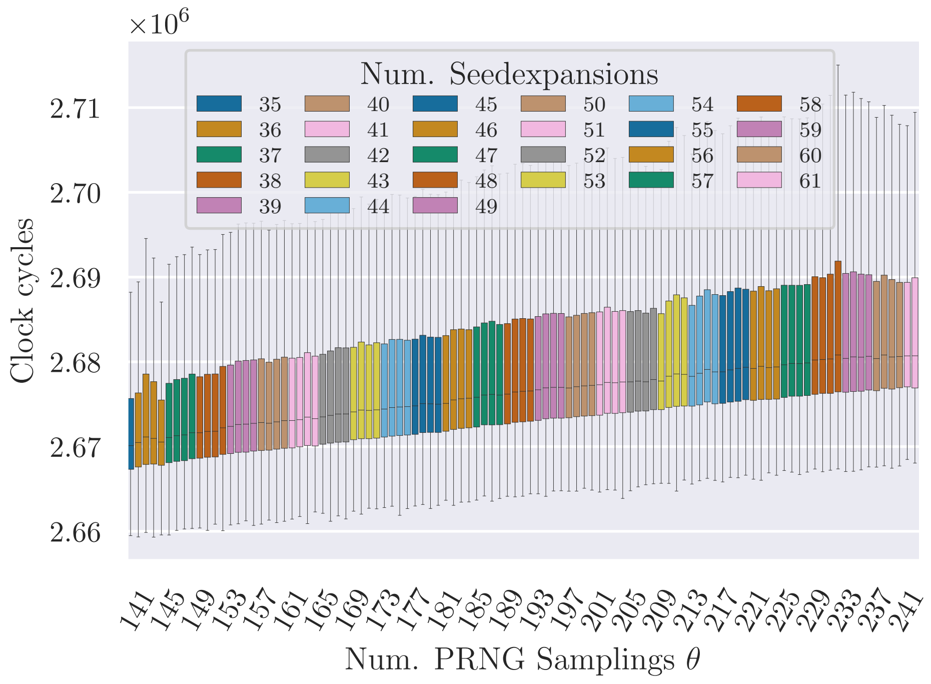


Figure 6: Box plots of 10^6 timing measurements of *BIKE-L1 Decaps* per value of θ. Values of θ with too few available plaintexts (< 10) are not simulated.

most important property to consider when determining the design and parameters of the distinguisher.

For the attack to succeed the probability of $|\theta_m - \theta_{m'}| < \Delta\theta$ must be minimized. This can be accomplished in two ways. First, by increasing the work-load of the pre-computation phase we can find a candidate value of m with θ_m as high/low as possible. The second way is to reduce the granularity ($\Delta\theta$) of the distinguisher by increasing the number of decapsulation measurements.

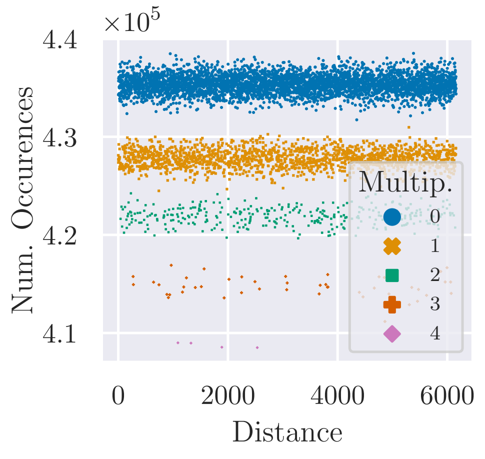
The probability of a distinguisher failure ϵ can easily determined by simulation. There are many ways to construct a distinguisher with a reasonably low failure rate ϵ. We selected a simple strategy where we use the minimum θ_m and where we use the 1% lowest measurement as the representative value for each distinguisher decision. This is done in order to select the value which is as close to noise free as possible.[8]

The distinguisher uses 2 phases; the profiling phase and the decision phase. In the profiling phase we first select (m, θ_m) and $\Delta\theta$ such that the the probability of $|\theta_m - \theta_{m'}| >= \Delta\theta$, for a random $\theta_{m'}$, is less than the targeted ϵ. Then a large number of measurements are collected for plaintext m^* where $\theta_{m^*} = \theta_m + \Delta\theta$. The 1% lowest measurement is selected as a threshold.

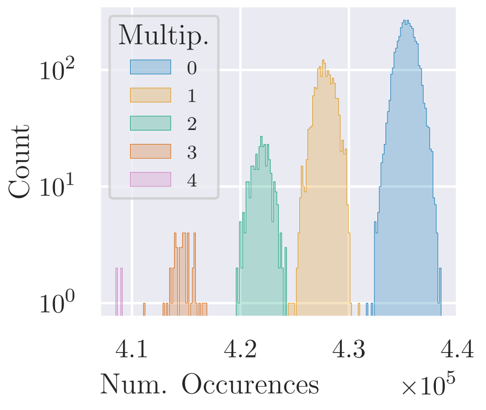
In the decision phase a number of decapsulation measurements are collected and, again, the 1% lowest value is selected. The selected value is compared against the previously selected threshold. If the measurement value is above the threshold we guess a decoding failure. If below, we determine decoding successful.

Clearly, the distinguisher can be made more sophisticated using statistical hypothesis testing or machine learning. However, for simply validating the practicality of exploiting

[8]We don't select the absolute minimum value as we have discovered that sometimes those values are impossible outliers. An hypothesis is that they come from instruction-reordering by the CPU and/or scheduling between CPU cores.



(a) Simulated aggregated distance spectrum using the *Error Amplification* attack. Listed in the legend is the various multiplicities $\mu(d)$ of the secret key.



(b) Plotting as histogram, the multiplicities $\mu(d)$ are quite separated and there should be no errors while determining the distance spectrum of the secret key.

Figure 7: Data is generated using $N_f = 8.5 \times 10^6$ decoding failures and a distinguisher with $\epsilon = 0.01$. For each distance is listed the number of occurrences that the specific distance was included in an error pattern that resulted in a decoding failure. The stratification into separate layers for each multiplicity is clearly visible.

this side-channel, this distinguisher is quite sufficient.

Simulations show that we can obtain $\epsilon \approx 0.01$ by constructing a distinguisher with $\Delta\theta = 22$. This value is obtained with a candidate plaintext m with $\theta_m = 138$ and $N_d = 1000$ decapsulation measurements, per decision. This was determined using the above parameters against 10^4 random error patterns of hamming weight 157, resulting in a DFR of 0.1369.

To complete the empirical evaluation of the attack we finally perform a full simulation of Algorithm 14. We have implemented the attack using an idealized oracle that output θ, the number of PRNG samplings performed by H. To show the real-world applicability of the attack the idealized oracle additionally simulates $\epsilon = 0.01$, artificially. The simulation results in the graphical representation of the distance spectrum of the secret key, as seen in Fig. 7.

Due to ϵ, a confirmation step was added where each found decoding failure was confirmed by an additional set of measurements. Otherwise the *Error Amplification* attack is sensitive to bad distinguisher decisions. Due to the chain of error patterns that is constructed it is critical that consecutive decoding failures are not missclassified. The extra confirmation step prevents this.

Observed in the figure is a clear picture of the distance spectrum without classification errors. This figure was obtained after about $N_f = 8.5 \times 10^6$ decoding failures in the decapsulation method for BIKE-L1. The simulation used a hamming weight of 149, which using the *Error Amplification* attack resulted in a DFR ≈ 0.146, a good match for our distinguisher above.

The final step is to do the key recovery, as detailed in [GJS16]. As in [GJS16], the reconstruction cost is negligible compared with the cost of querying the decryption oracles if the distance spectrum is fully recovered. The reason is that after quite few steps, the wrong guesses will be rejected with high probability and only the correct guess path will continue. One could balance the costs of building the distance spectrum and recovering the secret key, as done in [GJW19], by allowing errors in the recovered distance spectrum. As our aim is to demonstrate the new attack method, for simplicity, we leave this optimization trick for future works.

Finally, we estimate that to perform an actual key recovery we require about 6.7×10^{10} decapsulation measurements, or equivalently, 5.8×10^{7} number of idealized oracle calls. These numbers are given by:

$$\underbrace{\frac{N_f}{\text{DFR}}}_{\substack{\text{Ideal}\\ \text{oracle}}} \times N_d + \underbrace{N_f \times N_d}_{\text{Confirmation}} = N_d \times N_f \times \frac{1+\text{DFR}}{\text{DFR}}.$$

We firmly believe that these numbers can be reduced by, e.g., by an improvement of the distinguisher by statistical hypothesis testing, machine learning, or discarding ambiguous results, etc. Further options are optimizing the hamming weight (and thus the DFR) of the error patterns, allowing for larger ϵ and adding more confirmation steps, if necessary, allowing errors in the distance spectrum by trading for increased computational cost in the postprocessing stage, or spending further computational resources towards finding a more distinct (m, θ_m).

4.3 Message-Recovery from the New Weight Check

The described key-recovery timing attack on BIKE does not work for the new Github implementation and other related implementations (say in liboqs), due to an additional weight check on the error vector before the call to the H function, a check that is not explicitly specified. However, combined with the timing variation from the rejection sampling, this new weight check opens a new path for very efficient message recovery. In this section, we briefly describe such a message recovery attack.

We first build a distinguisher to distinguish if the weight check fails. In the new implementation, in Algorithm 8 after the call $\mathbf{e}' = \mathsf{Decode}()$ in line 1, the weight of $\mathbf{e}'$ is checked. If the weight is not t, $\mathbf{e}'$ is assigned a random value, otherwise it is kept. In line 2, $m' = c_1 \oplus \mathsf{L}(\mathbf{e}')$ and in the next line there is the call to $\mathsf{H}(m')$.

When we submit the same ciphertext to the BIKE decaps oracle multiple times, the input to the H function call are different random vectors if the weight check fails; otherwise, if the weight of $\mathbf{e}'$ is t, the inputs will be the same fixed vector. We can then build a distinguisher assuming that such execution time difference could be detected statistically after repeating the decaps oracle calls with the same ciphertext for many times.

With this distinguisher, we could launch a simple message-recovery attack. Assume that a correctly generated ciphertext $c = (\mathbf{c_0}, c_1) = (\mathbf{e_0} + \mathbf{e_1}\mathbf{h}, m \oplus \mathsf{L}(\mathbf{e_0}, \mathbf{e_1}))$ is received. One can now flip the first and the i^{th} bits of $\mathbf{c_0}$ (i.e., flipping the first and the i^{th} bits of unknown $\mathbf{e_0}$) and send the new ciphertext to the decaps oracle multiple times. Even though the decaps output is meaningless, from the distinguisher using the timing measurement one can detect if the flipped two bits have the same value. If they have the same value, the weight of $\mathbf{e}'$ will increase by 2, but if they have different values the weight of $\mathbf{e}'$ will be t. Since $\mathbf{e_0}$ is an extremely sparse vector, if we have more pairs among the $(r-1)$ pairs to be the same, the first bit of $\mathbf{e_0}$ should be 0; otherwise, it should be 1. When the first bit of $\mathbf{e_0}$ is decided, one can estimate the i^{th} bit of $\mathbf{e_0}$.

Note that one could also flip the first bit of $\mathbf{c_0}$ and add the vector $\mathbf{E_i}\mathbf{h}$ to $\mathbf{c_0}$, where $\mathbf{E_i}$ is the vector with only the i^{th} position nonzero. This is equivalent to flip the first bit of $\mathbf{e_0}$ and the i^{th} bit of $\mathbf{e_1}$. Thus, the value of $\mathbf{e_1}$ can be roughly estimated. We can then employ a post-processing step with ISD (information set decoding) algorithms to correct some distinguishing errors in the previous steps.

The attack was not implemented since the additional check is not part of the specification, but we can do a rough estimate of the complexity of such an attack. If we do a few hundred oracle calls with the same ciphertext and run through $2r$ different modified ciphertexts it appears very likely that the attack would be successful, since it would allow for some distinguishing errors that an ISD approach would then correct. In total, we may use several million oracle calls, which is much less than the described key-recovery attacks. We leave the investigation of its exact performance for future work. The conclusion is that the added weight check as it is done in the implementation is only weakening the security of the scheme.

5 Discussion on Countermeasures

To counter our proposed attacks and to remove the exploitable leakage, we see two ways: When analysing the construction of the KEM version of HQC, one might argue that finding a different tranformation from the IND-CPA PKE that eliminates the re-encryption step in the decapsulation might solve the problem. BIKE actually manages to optimize away the re-encryption step using the HHK implicit rejection transform, but still needs to retain the call to the constant-weight hashing funtion to check for decryption errors. Exploring different options for IND-CCA transformations in both schemes might be interesting future work. Short of finding a different IND-CCA transformation, the sampling of a fixed weight bit vector must be implemented isochronously.

We focus on a constant-time implementation of the algorithm. Since both attacks we presented exploit a structurally similar side-channel the countermeasure we present could be applied to both HQC and BIKE. We implemented and evaluated the countermeasure for HQC only. We identify two avenues for implementing a low fixed-weight vector sampling algorithm that is constant-time in the used seed. For the first one we initialize the vector of length n starting with a run of w set bits. Then we shuffle the array. This will result in a random vector of weight w. To shuffle the array one could use, e.g., the Fisher-Yates shuffling algorithm as described in [Knu97, p.145]. A naïve implementation of Fisher-Yates shuffling will leak timing information, as it will use secret-dependent array accesses. Using generic methods to make these array accesses constant-time results in an unacceptable asymptotic time complexity. Nicolas Sendrier presents a sophisticated approach how the Fisher-Yates shuffle can be modified to reduce the time complexity specifically for the use-case of generating random low fixed-weight vectors [Sen21]. For the case of BIKE, he shows that a small bias in the sampling distribution has a negligible impact on the security of the scheme. His work was published as a response to the initial pre-print version of our work, which only contained the attack on HQC and its countermeasure. While we are not aware of an implementation and concrete evaluation, the decreased time complexity of Sendrier's approach makes the method very compelling. Our approach requires little deviation from the already existing code-base, and allows existing implementations to be patched with relative ease. Another approach would be to use a reverse sorting algorithm, using an established sorting algorithm like merge-sort, as it is proposed in [WSN18] for a Classic McEliece hardware implementation. The reverse merge-sort induces a slight bias which is solved by a rejection and is therefore not suitable for a constant-time implementation, unless one can show that the bias is acceptable. The Beneŝ-network used in the C reference implementation of Classic McEliece is aligned to a vector size of a power of 2 which is not the case in HQC.

We propose an approach that samples the specified number of distinct bit positions in constant-time and sets the bits in the vector in constant-time. For HQC, only the distinct position sampling is not constant-time. Our modification results in an algorithm that is only probabilistically correct and may sample too few distinct bit positions. The probability of this failure mode can be chosen arbitrarily small and made negligible.

To obtain our final countermeasure we perform several modifications to the algorithm. After each modification the algorithm is a fully functioning algorithm, however has different

```
 void vect_set_random_fixed_weight(
   seedexpander_state *ctx,
   __m256i *v256, uint16_t weight) {
-  size_t random_bytes_size = 3 * weight;
+  size_t random_bytes_size = 2 * 3 * weight;
-  uint8_t rand_bytes[3 * PARAM_OMEGA_R] = {0};
+  uint8_t rand_bytes[2 * 3 * PARAM_OMEGA_R] = {0};
```

Figure 8: 4 byte patch to HQC to remove additional `seedexpander` calls

side-channel behavior. The first modification in Section 5.1 is the simplest modification possible: a 4 byte patch that prevents the `seedexpander` function from being called a varying number of times in a single Sample call. It works by finding a loose upper bound for the number of bytes that the entire Sample function requires. Following, in Section 5.2 we replace a rejection sampling algorithm used to generate random positions in the range $\{0, \ldots, n-1\}$ with a constant time algorithm using modulo reduction of a large number. In Section 5.3 we then detail how the loose upper bound for the number of bytes to sample used in Section 5.3 can be tightened by accurately modeling the distribution of the number of times a number is sampled. We compute parameters required to make the probability that the number of bytes sampled is insufficient negligible. The resulting parameters depend upon whether the countermeasure in Section 5.2 was applied (κ_1) or not (κ_{p_s}), as the original version can fail to sample a number (in the case of a rejection) and our countermeasure always succeeds. As a final modification in Section 5.4 we modify the the loop that generates random distinct positions to always perform the same number of iterations κ resulting in a constant-time algorithm that can fail to produce a result of the expected weight with negligible probability. The algorithm is approximately correct, because we always perform κ iterations, even if we would need more iterations because e.g. many bit positions collided with previously sampled ones. Lastly, we benchmark the resulting countermeasure and compare it to the original in Section 5.5.

5.1 Remove Additional `seedexpander` Calls

The first attempt we make to get a countermeasure is to eradicate the concrete side-channel that we use for the attack; The rejection sampling algorithm generates new random data using the `seedexpander` function on demand. It is vanishingly unlikely that a single Sample invocation induces more than one additional `seedexpander` call. Therefore, our first, obvious countermeasure is to increase the number of bytes that are generated initially to double the previous amount. This results in a patch to the sampling function shown in Fig. 8. However, the algorithm is not constant-time: rejection sampling still performs a different number of iterations depending on the message and each random number is also sampled using rejection-sampling. While the countermeasure increases the effort required to perform the attack, it could still permit attacks in a low-noise environment to recover the key. Further, for BIKE the number of seedexpansions is not as large a factor for the timing distribution and therefore such a patch would not have a lot of impact for BIKE.

5.2 On Constant-Time Random Number Generation

To further reduce the timing leakage we can try to remove the inner rejection used for generating random indices into the vector. The inner rejection sampling is detailed in Algorithm 16. Instead of rejection sampling integers in the range $0 \leq x < \lfloor 2^k/m \rfloor m$, we generate $b \gg \log_2 m$ random bits and then reduce the generated integer modulo m to the desired range. This will bias the resulting distribution if m does not divide 2^b, which is

Algorithm 16: Inner Rejection Sampling Algorithm

```
Result: Random number in $[0, \dots, m-1]$
1 repeat
2     $i \leftarrow\$ [0, 2^k)$
3 until $i < \left\lfloor \frac{2^k}{m} \right\rfloor m$
4 return $i \bmod m$
```

the case here. Therefore, we need to pick a sufficiently large b for the statistical distance to be negligible. In particular we are interested in minimizing the statistical distance (SD) between the uniform distribution over $\{0, \dots, m-1\}$ and the distribution generated by $x \bmod m$ where x is drawn uniformly random random from $\{0, \dots, 2^b - 1\}$. We define the statistical distance between two probability distributions X and Y over some discrete domain Ω to be:

$$\mathrm{SD}_{X,Y} = \frac{1}{2} \cdot \sum_{z \in \Omega} |\Pr[X = z] - \Pr[Y = z]|$$

Let U_m be the uniform probability distribution over $\{0, \dots, m-1\}$:

$$\Pr[U_m = z] = \begin{cases} \frac{1}{z} & \text{if } 0 \le z < m \\ 0 & \text{otherwise} \end{cases}$$

Additionally, we define the probability distribution M_n which reduces an integer in $\{0, \dots, n-1\}$ modulo m. Its probability distribution is given by:

$$\Pr[M_n = z] = \begin{cases} \frac{\lfloor n/m \rfloor + 1}{n} & \text{if } 0 \le z < n \bmod m \\ \frac{\lfloor n/m \rfloor}{n} & \text{if } n \bmod m \le z < m \\ 0 & \text{otherwise} \end{cases}$$

The statistical distance between these two distributions is:

$$\mathrm{SD}_{U_m, M_n} = \frac{1}{2} \cdot \sum_{z \in \{0, \dots, m-1\}} |\Pr[U_m = z] - \Pr[M_b = z]| \tag{7}$$

$$= \frac{1}{2} \cdot \left((n \bmod m) \cdot \left| \frac{1}{m} - \frac{\lfloor \frac{n}{m} \rfloor + 1}{n} \right| + \right. \tag{8}$$

$$\left. (m - (n \bmod m)) \cdot \left| \frac{1}{m} - \frac{\lfloor \frac{n}{m} \rfloor}{n} \right| \right)$$

In Table 5 we computed the statistical distance between the uniform distribution and the modular reduction technique for various numbers of bits b. The parameter m is the length of the vector in HQC. We leave the choice of an acceptable statistical distance to the designers of the scheme. For our further testing we use $b = 128$.

We can implement a modular reduction of a b-bit non-negative number x modulo a small number efficiently using basic rules of modular arithmetic. We can represent x in e.g. base 2^8 as $x = x_0 + 2^8 \cdot x_1 + 2^{8 \cdot 2} \cdot x_2 + \cdots + 2^{8 \cdot (\ell - 1)} x_{\ell-1} + 2^{8 \cdot \ell} \cdot x_\ell$ where $\ell = \lceil \frac{b}{8} \rceil$. We split up the computation of $x \bmod m$ in the following way:

$$x \bmod m = \left(\cdots \overbrace{\left(x_{\ell-1} + 2^8 \cdot (\underbrace{x_\ell \bmod m}_{z_0}) \right) \bmod m}^{z_1} \cdots \right) \bmod m$$

Table 5: Statistical distance between the uniform distribution over $\{0, \ldots, m-1\}$ and the distribution of random integers from 0 to $2^b - 1$ reduced modulo m for hqc-128 ($n = 17{,}669$).

b	$\log_2 \mathrm{SD}_{U_m, M_{2^b}}$ (approx.)
16	-4
32	-20
64	-52
128	-116
256	-244
512	-502

```
uint32_t random_data = 0;
for (uint32_t k = 0; k < BYTES_PER_INDEX; ++k) {
  random_data = ((uint32_t)rand_bytes[j++] | (random_data << 8));
  random_data %= PARAM_N;
}
```

Figure 9: Constant time modulo reduction of $x \bmod m$ in multiple steps where `BYTES_PER_INDEX` is $\lceil \frac{b}{8} \rceil$.

Generalizing this, we can write an iterative algorithm that computes in iteration i:

$$z_i = \begin{cases} x_\ell \bmod m & \text{if } i = 0 \\ (x_{\ell-i} + 2^8 \cdot z_{i-1}) \bmod m & \text{otherwise} \end{cases}$$

and $z_\ell = x \bmod m$. We can implement this algorithm for a random number x where each x_i is drawn from `rand_bytes` as shown in Fig. 9.

Additionally, while a divide instruction is not constant-time in general on most Instruction Set Architectures (ISAs), reducing modulo a constant is optimized by the compiler into a sequence of instructions that can be executed in constant time. The optimization performed by the compiler is a Barrett reduction [MvOV97, p.603]. This can be observed in Fig. 10. Here the compiler replaced the `idiv` instruction by a series of shifts, additions and multiplications. All of these instructions complete with a fixed latency on the Zen 3 ISA according to Agner's instruction tables[9]. To ensure that the compiled result always uses these instructions, which we have verified to be constant-time, we can copy the compilation result into an `__asm__ volatile` block.

```
#include <stdint.h>

uint32_t f(uint32_t a) {
    return a % 23869;
}
```

```
f:
    mov eax, edi
    mov ecx, edi
    mov edx, 2948122845
    imul    rdx, rcx
    shr rdx, 46
    imul    ecx, edx, 23869
    sub eax, ecx
    ret
```

Figure 10: Modular reduction of an integer a modulo a constant in C and the resulting Intel-style x86 assembly with optimization level 2 using clang.

[9] https://www.agner.org/optimize/instruction_tables.pdf, accessed on 2021-11-05.

5.3 Tight Upper Bound on the Number of Samples Required

We wish to minimize the number of random bytes generated, while still ensuring that we never run out of randomness during the rejection sampling function so that we only have to perform seedexpansion once at the beginning of the function. To this end, we analyze the probability of requiring a certain number of iterations in the rejection sampling algorithm. We introduce the random variable X_{n,i,p_s}, which is the number of distinct elements after attempting to sample i elements from $\{1,\ldots,n\}$ with each sample succeeding with the probability p_s. The success probability p_s can be used to model the case where the inner rejection sampling algorithm has to retry sampling an element from $\{1,\ldots,n\}$, because the sampled element was not in the required range. Therefore, if a sample fails, it increases the number of iterations, but no element is sampled. This yields the following recursive relation:

$$\Pr[X_{n,i,p_s}=w]=\begin{cases}0 & \text{if } i<w\\ 1 & \text{if } w=i=0\\ p_s & \text{if } w=i=1\\ \begin{array}{r} p_s\dfrac{w}{n}\Pr[X_{n,i-1,p_s}=w]+\\ (1-p_s\max(0,\dfrac{w-1}{n}))\Pr[X_{n,i-1,p_s}=w-1]\end{array} & \text{otherwise}\end{cases} \tag{9}$$

We are now sufficiently equipped to compute the probability that the rejection sampling algorithm requires $\leq i$ iterations to sample w distinct bit positions. This query is equivalent to the probability, that after i iterations $\geq w$ distinct bit positions have been sampled. We can compute this by simply summing over the number of distinct positions:

$$\Pr[X_{n,i,p_s}\geq w]=\sum_{x=w}^{i}\Pr[X_{n,i,p_s}=x]$$

Finally, we define the random variable U_{n,w,p_s} to be the number of iterations required to sample w distinct elements out of $\{1,\ldots,n\}$ with each sample succeeding with probability p_s. Then, the probability of requiring $\leq i$ iterations is:

$$\Pr[U_{n,w,p_s}\leq i]=\Pr[X_{n,i,p_s}\geq w]$$

We can use the random variable U_{n,w,p_s} to minimize the number of random bytes that we need to sample. The probability that a message emits ≥ 1 additional `seedexpander` calls when the randomness reservoir provides sufficient entropy for κ random indices is:

$$1-(\Pr[U_{n,\omega_r,p_s}\leq\kappa])^3.$$

We would like this probability to be negligible. We can compute a suitable κ for which the probability is $\leq 2^{-\lambda}$ where λ is the security parameter. This is done by increasing κ until the probability is low enough. The number of iterations depends on the success probability of sampling a random index. When we retain the original inner rejection sampling algorithm we use the success probability p_s to compute κ_{p_s}. For the constant-time random number generation we use a success probability of 1 to compute κ_1. Note that these probabilities are high enough for these messages to feasibly exist. However, we deem it infeasible to compute such messages, as they are so rare.

The results of these computations can be seen in Table 6. Using κ we can optimize the countermeasure to generate the least amount of randomness to eradicate additional `seedexpander` calls. Note that $\kappa_1\leq\kappa_{p_s}$, since the rejection sampling algorithm requires less iterations when every random number generation succeeds. However, the constant-time

Table 6: Number of indices κ that must be derivable from the generated randomness reservoir to achieve a probability on the order of the security parameter of a message emitting multiple `seedexpander` calls. Here, p_s is set to $\lfloor 2^k/m \rfloor m/2^k$ for when the original rejection sampling is used or 1 when the bit position sampling cannot fail due to the use of the constant-time random number generation scheme.

Instance	κ_{p_s}	$\log_2(1-(\Pr[U_{n,\omega_r,p_s} \le \kappa_{p_s}])^3)$	κ_1	$\log_2(1-(\Pr[U_{n,\omega_r,1} \le \kappa_1])^3)$
hqc-128	99	≈ -134	97	≈ -129
hqc-192	152	≈ -193	146	≈ -195
hqc-256	192	≈ -261	190	≈ -259

Random Number Generator (RNG) still requires much more random bytes to be generated, since it requires 16 bytes per index, instead of approx. 3 in expectation.

We can further optimize the runtime of the RNG by using the full width of the ISA's registers. Instead of reducing one byte at a time we can reduce 4 bytes at once by using 64 bit registers and multiplying each intermediate result z_{i-1} by $2^{8\cdot 4}$, as we detail in Listing 1. Further performance improvements may be achievable through the use of even wider registers or SIMD instructions to produce multiple positions at once.

5.4 Constant-Time Monte-Carlo

We can now forge a constant-time algorithm that is approximately correct using minimal modifications. It fails to produce a correct result with an error-probability that we can choose to be arbitrarily low. The first step is to always produce the same number of random positions into the generated vector. Additionally, for each position we keep track of whether it is needed, i.e., whether the generated index has already been sampled before and whether we have already sampled enough unique indices. Using this information, we can then set the bit only if it is needed – in constant time. However, if we fail to sample enough unique indices, the algorithm may produce a vector of too low weight. We cannot catch this error and try again, as that would introduce a timing-variability. Therefore, we must sample enough positions such that this case does not happen with overwhelming probability. We can reuse the κ_1 listed in Table 6 for this purpose. Using these parameters the probability that we sample a vector of too low weight is $\le 2^{-\lambda}$, where λ is the security parameter.

Concretely, we keep track of the number of unique positions sampled and whether we need each position by:

```
uint32_t count = 0;
uint8_t take[K_1];
```

We then sample κ_1 positions from $\{0, \ldots, n-1\}$. Instead of trying to sample a position again when a position is not unique, we store it unconditionally but keep track of whether

```
uint32_t rand_bytes[BYTES_PER_INDEX * K_1 / 4] = {0};
// [...]
uint64_t random_data = 0;
for (uint32_t k = 0; k < BYTES_PER_INDEX / 4; ++k) {
  random_data = (uint64_t) rand_bytes[j++] + (random_data << 32);
  random_data %= PARAM_N;
}
```

Listing 1: Optimization in the random number generation by reducing 4 bytes at once.

we need the position:

```
tmp[i] = random_data;
uint8_t not_enough = count < weight;
uint8_t needed = (!exist) & not_enough;
take[i] = needed;
count += needed;
```

where `exist` is `1` iff the position has not been sampled before and `i` is the iteration count in $\{0, \ldots, \kappa_1 - 1\}$. To avoid naming ambiguities in this section we henceforth refer to the vector of n bits that is modified by the algorithm as the *bit-array*.

The next phase of the algorithm uses Advanced Vector Extensions (AVX2) instructions to set the sampled bit positions in the bit-array. This algorithm is vectorized to process the bit-array in 256 bit chunks. We modify this algorithm to only include a position if `take[i]` is set by computing a bit mask that is 1^{256} if `take[i] == 1` and 0^{256} otherwise. We then modify the first loop to compute the bitwise and of `bit256[i]` and the mask stored in `take256`:

```
__m256i take256 = _mm256_set1_epi64x(take[i]) == 1;
bit256[i] = bloc256&mask256&take256;
```

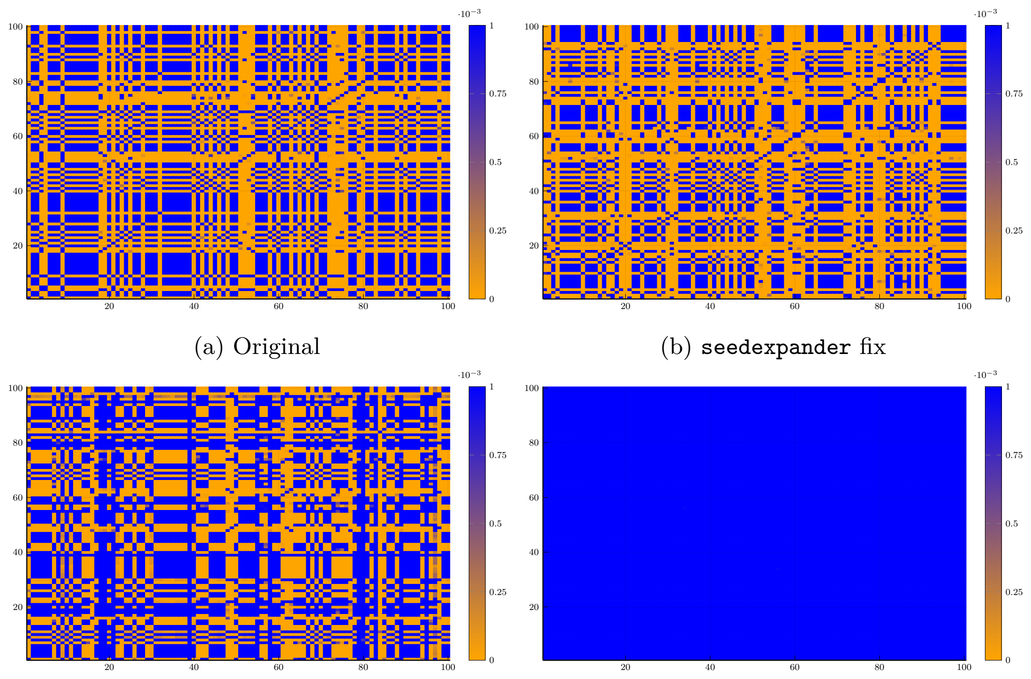
This results in `bit256[i]` being 0^{256} if the bit is not needed. When this 256 bit vector is later xor-ed with the `aux` variable, it will have no impact, since $0 \oplus x = x$.

5.5 Evaluation of the Proposed Countermeasures

The side-channel evaluation results can be viewed in Fig. 11. The number of the detected difference clearly diminishes as more of the suggested modifications are applied. In particular, the final countermeasure eradicates all statistically significant timing differences in the Sample function as can be seen in Fig. 11d. We conclude that the final countermeasure eradicates all timing-leakage that we could detect from the algorithm with respect to the seed used by the XOF.

We measure the number of cycles the Sample function requires for random messages for the original and the two patched versions to evaluate the performance impact of the additional instructions. We obtained 1 million measurements and removed outliers that deviate more than 3 standard deviations from the mean. Additionally, we gave the process a niceness of −20 on a dedicated machine. The process is pinned to a single core, and all other processes are pinned to different cores. The results may be seen in Table 7. We collected the mean and median number of cycles. The median number of cycles is increasing with more patches applied. We can see that the RNG fix is extremely costly in terms of cycle count and together with the `seedexpander` fix induces a 22.8 % increase in the median number of cycles. The main fault is likely that the constant-time RNG method generates and processes approximately 5 times the number of random bytes. Furthermore, we observe that the `seedexpander` patch alone is extremely cheap and only incurs a 1 % increase in the number of cycles.

While fixing the `seedexpander` side-channel is cheap, it is not sufficient to obtain constant-time code. We were able to use the constant-time RNG in the design of further algorithms. Unfortunately, the constant-time RNG comes with a heavy performance hit, and it is not trivial to decide on a number of bytes to consume for each generated position. The final modification is constant-time, however it has a non-zero probability of returning an incorrect result. We choose this probability low enough for this to likely not be a practical issue.



(a) Original

(b) `seedexpander` fix

(c) `seedexpander` + RNG fix

(d) `seedexpander` + RNG fix + constant iters

Figure 11: P-values for Welch's t-test testing whether there is a statistically significant difference between the computation time of the part invoking the Sample function in the re-encryption of the decapsulation for each pair in 100 ciphertexts generated for a single keypair. Orange indicates that a statistically significant difference was detected. The final countermeasure eradicates all statistically significant timing differences in the Sample function.

Table 7: Benchmark results in number of cycles for the modifications of the rejection sampling algorithm. Modifications tested on hqc-128. Cycle counts include the entirety of the decapsulation function.

Version	Median Cycles
original	259,370 (+ 0.0%)
`seedexpander` fix	261,849 (+ 1.0%)
`seedexpander` + RNG fix	318,533 (+22.8%)
`seedexpander` + RNG fix + constant number of iterations	334,628 (+29.0%)

6 Conclusions, Lessons, and Future Work

We have presented novel key-recovery timing attacks on HQC and BIKE, where non-constant-time rejection sampling procedures are implemented for generating random vectors with a specific weight. The time differences caused by rejection sampling could leak whether the input message to the deterministic re-encryption procedure (or to a hash function) in the IND-CCA transformation is unchanged. Such secret information is sufficient for recovering the secret key of HQC and BIKE.

The considered implementation of HQC in this work has been found vulnerable despite the claim of the authors of HQC that the recent code is thoroughly analyzed so that only unused randomness (i.e., rejected based on public criteria) or otherwise nonsensitive data is leaked. The identified vulnerability probably has been hidden from scrutiny because the modular design of the HQC KEM employing the FO transformation conceals the dependence of the secret key to the rejection sampling function, due to a subtle error in the specification: In the IND-CPA version of HQC, encryption is non-deterministic, and thus the variations of the employed rejection sampling function is of no concern. The KEM/DEM version of HQC, as specified in Figure 3 in the specification, invokes a slightly different HQC.PKE encryption scheme than the one described in Figure 2 of the specification: one that fixes the randomness to make encryption deterministic. Only because the re-encryption in the KEM decapsulation is deterministic and because the seed is derived from secret data, non-constant-time rejection sampling becomes a problem. This highlights the issue of providing high level definitions of a cryptosystem: The definition is good enough for an implementer to get the functionality correct, but hides from manual inspection the ominous dependence identified and exploited in this work. However, in the case of HQC the specification encourages the use of the exploited rejection sampling algorithm. Therefore, the flaw we identify would likely be implemented by any implementer. This problem also highlights the need for automated, possibly standardized tools to check implementations for secret-dependent timing variations.

Regarding BIKE, we have identified a timing variation very similar to the one discovered in HQC. This vulnerability can be exploited for a key-recovery attack on BIKE. We found several vulnerable implementations, including the implementations in the NIST round 3 submission. Interestingly, in the most recent versions from Github, an additional weight check before the re-encryption procedure is employed, which can make the current key-recovery attack version unpractical. We emphasize that this additional weight check actually weakens the security of BIKE, since such a weight check allows more efficient message-recovery attacks. We still suggest to have a fully constant-time implementation of BIKE.

Our proposed countermeasure does incur a heavy performance degradation. However, it does eliminate all timing-variations that we could detect from the analyzed function. The constant-time variant of the Fisher-Yates algorithm proposed by Sendrier in a parallel work to ours introduces a slight bias in the uniform distribution but without an impact on the security properties of the scheme. It is another very interesting approach and should be considered in upcoming implementations and research activities as well. Future work could focus on improving the performance of the mentioned countermeasures through the use of SIMD instructions or different algorithms.

References

[AAB$^+$] Carlos Aguilar Melchor, Nicolas Aragon, Slim Bettaieb, Loïc Bidoux, Olivier Blazy, Jean-Christophe Deneuville, Philippe Gaborit, Edoardo Persichetti, Gilles Zémor, and Jurjen Bos. Optimized implementation of HQC. available at:https://pqc-hqc.org/download.php?file=hqc-optimized-implementation_2021-06-06.zip.

[AAB+21] Carlos Aguilar Melchor, Nicolas Aragon, Slim Bettaieb, Loïc Bidoux, Olivier Blazy, Jean-Christophe Deneuville, Philippe Gaborit, Edoardo Persichetti, Gilles Zémor, and Jurjen Bos. HQC. Technical report, National Institute of Standards and Technology, 2021. available at https://csrc.nist.gov/projects/post-quantum-cryptography/round-3-submissions.

[ABB+20] Nicolas Aragon, Paulo Barreto, Slim Bettaieb, Loic Bidoux, Olivier Blazy, Jean-Christophe Deneuville, Philippe Gaborit, Shay Gueron, Tim Guneysu, Carlos Aguilar Melchor, Rafael Misoczki, Edoardo Persichetti, Nicolas Sendrier, Jean-Pierre Tillich, Gilles Zémor, Valentin Vasseur, and Santosh Ghosh. BIKE. Technical report, National Institute of Standards and Technology, 2020. available at https://csrc.nist.gov/projects/post-quantum-cryptography/round-3-submissions.

[ABB+21] Nicolas Aragon, Paulo Barreto, Slim Bettaieb, Loic Bidoux, Olivier Blazy, Jean-Christophe Deneuville, Philippe Gaborit, Shay Gueron, Tim Guneysu, Carlos Aguilar Melchor, Rafael Misoczki, Edoardo Persichetti, Nicolas Sendrier, Jean-Pierre Tillich, Gilles Zemor, Valentin Vasseur, Santosh Ghosh, and Jan Richter-Brokmann. BIKE. Technical report, National Institute of Standards and Technology, 2021. available at https://csrc.nist.gov/projects/post-quantum-cryptography/round-3-submissions.

[AIES15] Gorka Irazoqui Apecechea, Mehmet Sinan Inci, Thomas Eisenbarth, and Berk Sunar. Lucky 13 strikes back. In Feng Bao, Steven Miller, Jianying Zhou, and Gail-Joon Ahn, editors, *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security, ASIA CCS '15, Singapore, April 14-17, 2015*, pages 85–96. ACM, 2015.

[AMBD+18] Carlos Aguilar-Melchor, Olivier Blazy, Jean Christophe Deneuville, Philippe Gaborit, and Gilles Zemor. Efficient Encryption from Random Quasi-Cyclic Codes. *IEEE Transactions on Information Theory*, 64(5):3927–3943, 2018.

[AP13] Nadhem J. AlFardan and Kenneth G. Paterson. Lucky thirteen: Breaking the TLS and DTLS record protocols. In *2013 IEEE Symposium on Security and Privacy, SP 2013, Berkeley, CA, USA, May 19-22, 2013*, pages 526–540. IEEE Computer Society, 2013.

[BB05] David Brumley and Dan Boneh. Remote timing attacks are practical. *Computer Networks*, 48(5):701–716, 2005.

[BDH+19] Ciprian Băetu, F. Betül Durak, Loïs Huguenin-Dumittan, Abdullah Talayhan, and Serge Vaudenay. Misuse attacks on post-quantum cryptosystems. In Yuval Ishai and Vincent Rijmen, editors, *EUROCRYPT 2019, Part II*, volume 11477 of *LNCS*, pages 747–776. Springer, Heidelberg, May 2019.

[BDL97] Dan Boneh, Richard A. DeMillo, and Richard J. Lipton. On the importance of checking cryptographic protocols for faults (extended abstract). In Walter Fumy, editor, *Advances in Cryptology - EUROCRYPT '97, International Conference on the Theory and Application of Cryptographic Techniques, Konstanz, Germany, May 11-15, 1997, Proceeding*, volume 1233 of *Lecture Notes in Computer Science*, pages 37–51. Springer, 1997.

[BDL01] Dan Boneh, Richard A. DeMillo, and Richard J. Lipton. On the importance of eliminating errors in cryptographic computations. *Journal of Cryptology*, 14(2):101–119, 2001.

[Ben17] Simon Benjamin. Perspectives on the State of Affairs for Scalable Fault-Tolerant Quantum Computers and Prospects for the Future. Presented at the 5th ETSI-IQC Workshop on Quantum-Safe Cryptography, 2017. available at `https://docbox.etsi.org/Workshop/2017/201709_ETSI_IQC_QUANTUMSAFE/TECHNICAL_TRACK/S03_THREATS/UNIofOXFORD_BENJAMIN.pdf`.

[BHLY16] Leon Groot Bruinderink, Andreas Hülsing, Tanja Lange, and Yuval Yarom. Flush, gauss, and reload - A cache attack on the BLISS lattice-based signature scheme. In Benedikt Gierlichs and Axel Y. Poschmann, editors, *CHES 2016*, volume 9813 of *LNCS*, pages 323–345. Springer, Heidelberg, August 2016.

[BL16] Daniel J Bernstein and Tanja Lange. Failures in NIST's ECC standards. pages 1–27, 2016. available at: `https://cr.yp.to/newelliptic/nistecc-20160106.pdf`.

[BT11] Billy Bob Brumley and Nicola Tuveri. Remote timing attacks are still practical. In Vijay Atluri and Claudia Díaz, editors, *Computer Security - ESORICS 2011 - 16th European Symposium on Research in Computer Security, Leuven, Belgium, September 12-14, 2011. Proceedings*, volume 6879 of *Lecture Notes in Computer Science*, pages 355–371. Springer, 2011.

[CCK21] Ming-Shing Chen, Tung Chou, and Markus Krausz. Optimizing BIKE for the intel haswell and ARM cortex-M4. *IACR TCHES*, 2021(3):97–124, 2021. `https://tches.iacr.org/index.php/TCHES/article/view/8969`.

[CWR09] Scott A. Crosby, Dan S. Wallach, and Rudolf H. Riedi. Opportunities and limits of remote timing attacks. *ACM Trans. Inf. Syst. Secur.*, 12(3):17:1–17:29, 2009.

[DKA$^+$14] Zakir Durumeric, James Kasten, David Adrian, J. Alex Halderman, Michael Bailey, Frank Li, Nicholas Weaver, Johanna Amann, Jethro Beekman, Mathias Payer, and Vern Paxson. The matter of heartbleed. In Carey Williamson, Aditya Akella, and Nina Taft, editors, *Proceedings of the 2014 Internet Measurement Conference, IMC 2014, Vancouver, BC, Canada, November 5-7, 2014*, pages 475–488. ACM, 2014.

[Dya18] Mikhail Dyakonov. The Case Against Quantum Computing. IEEE Spectrum, 2018. available at: `https://spectrum.ieee.org/the-case-against-quantum-computing`.

[Gab21] Philippe Gaborit. Personal communication, November 2021.

[Gal62] Robert G. Gallager. Low-density parity-check codes. *IRE Trans. Inf. Theory*, 8(1):21–28, 1962.

[GJN20] Qian Guo, Thomas Johansson, and Alexander Nilsson. A key-recovery timing attack on post-quantum primitives using the Fujisaki-Okamoto transformation and its application on FrodoKEM. In Daniele Micciancio and Thomas Ristenpart, editors, *CRYPTO 2020, Part II*, volume 12171 of *LNCS*, pages 359–386. Springer, Heidelberg, August 2020.

[GJS16] Qian Guo, Thomas Johansson, and Paul Stankovski. A key recovery attack on MDPC with CCA security using decoding errors. In Jung Hee Cheon and Tsuyoshi Takagi, editors, *ASIACRYPT 2016, Part I*, volume 10031 of *LNCS*, pages 789–815. Springer, Heidelberg, December 2016.

[GJW19] Qian Guo, Thomas Johansson, and Paul Stankovski Wagner. A key recovery reaction attack on QC-MDPC. *IEEE Trans. Inf. Theory*, 65(3):1845–1861, 2019.

[HGS99] Chris Hall, Ian Goldberg, and Bruce Schneier. Reaction attacks against several public-key cryptosystems. In Vijay Varadharajan and Yi Mu, editors, *ICICS 99*, volume 1726 of *LNCS*, pages 2–12. Springer, Heidelberg, November 1999.

[HHK17] Dennis Hofheinz, Kathrin Hövelmanns, and Eike Kiltz. A modular analysis of the Fujisaki-Okamoto transformation. In Yael Kalai and Leonid Reyzin, editors, *TCC 2017, Part I*, volume 10677 of *LNCS*, pages 341–371. Springer, Heidelberg, November 2017.

[HLS21] Clemens Hlauschek, Norman Lahr, and Robin Leander Schröder. On the timing leakage of the deterministic re-encryption in hqc kem. Cryptology ePrint Archive, Report 2021/1485, version 20211115:124514 (posted 1636980314 15-Nov-2021 12:45:14 UTC), 8 2021. https://eprint.iacr.org/2021/1485/20211115:124514.

[Hog15] Mél Hogan. Data flows and water woes: The Utah Data Center. *Big Data & Society*, 2(2), 2015.

[HPA21] James Howe, Thomas Prest, and Daniel Apon. SoK: How (not) to design and implement post-quantum cryptography. In Kenneth G. Paterson, editor, *Topics in Cryptology – CT-RSA 2021*, pages 444–477, Cham, 2021. Springer International Publishing.

[Kal20] Gil Kalai. The argument against quantum computers, the quantum laws of nature, and google's supremacy claims. *CoRR*, abs/2008.05188, 2020.

[KJJ99] Paul Kocher, Joshua Jaffe, and Benjamin Jun. Differential Power Analysis. In *CRYPTO 1999*, pages 388–397. Springer US, Boston, MA, 1999.

[Knu97] Donald Ervin Knuth. *The art of computer programming, Volume I: Fundamental Algorithms, 3rd Edition.* Addison-Wesley, 1997.

[Koc96] Paul C. Kocher. Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems. In Neal Koblitz, editor, *Advances in Cryptology - CRYPTO '96, 16th Annual International Cryptology Conference, Santa Barbara, California, USA, August 18-22, 1996, Proceedings*, volume 1109 of *Lecture Notes in Computer Science*, pages 104–113. Springer, 1996.

[KPVV16] Thierry Kaufmann, Hervé Pelletier, Serge Vaudenay, and Karine Villegas. When constant-time source yields variable-time binary: Exploiting curve25519-donna built with MSVC 2015. In Sara Foresti and Giuseppe Persiano, editors, *Cryptology and Network Security - 15th International Conference, CANS 2016, Milan, Italy, November 14-16, 2016, Proceedings*, volume 10052 of *Lecture Notes in Computer Science*, pages 573–582, 2016.

[LLJ+19] Xianhui Lu, Yamin Liu, Dingding Jia, Haiyang Xue, Jingnan He, Zhenfei Zhang, Zhe Liu, Hao Yang, Bao Li, and Kunpeng Wang. LAC. Technical report, National Institute of Standards and Technology, 2019. available at https://csrc.nist.gov/projects/post-quantum-cryptography/round-2-submissions.

[MAA+20] Dustin Moody, Gorjan Alagic, Daniel C Apon, David A Cooper, Quynh H Dang, John M Kelsey, Yi-Kai Liu, Carl A Miller, Rene C Peralta, Ray A Perlner, Angela Y Robinson, Daniel C Smith-Tone, and Jacob Alperin-Sheriff. Status report on the second round of the NIST post-quantum cryptography standardization process. Technical report, National Institute of Standards and Technology, Gaithersburg, MD, jul 2020.

[MBA+21] Robert Merget, Marcus Brinkmann, Nimrod Aviram, Juraj Somorovsky, Johannes Mittmann, and Jörg Schwenk. Raccoon attack: Finding and exploiting most-significant-bit-oracles in TLS-DH(E). In Michael Bailey and Rachel Greenstadt, editors, *30th USENIX Security Symposium, USENIX Security 2021, August 11-13, 2021*, pages 213–230. USENIX Association, 2021.

[Mos17] Michele Mosca. The Quantum Threat to Cybersecurity (for CxOs). Presented at the 5th ETSI-IQC Workshop on Quantum-Safe Cryptography, 2017. available at https://docbox.etsi.org/Workshop/2017/201709_ETSI_IQC_QUANTUMSAFE/EXECUTIVE_TRACK/UNIofWATERLOO_MOSCA.pdf.

[MSEH20] Daniel Moghimi, Berk Sunar, Thomas Eisenbarth, and Nadia Heninger. TPM-FAIL: TPM meets timing and lattice attacks. In Srdjan Capkun and Franziska Roesner, editors, *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020*, pages 2057–2073. USENIX Association, 2020.

[MTSB13] Rafael Misoczki, Jean-Pierre Tillich, Nicolas Sendrier, and Paulo S. L. M. Barreto. Mdpc-mceliece: New mceliece variants from moderate density parity-check codes. In *Proceedings of the 2013 IEEE International Symposium on Information Theory, Istanbul, Turkey, July 7-12, 2013*, pages 2069–2073. IEEE, 2013.

[MvOV97] Alfred Menezes, Paul C. van Oorschot, and Scott A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 1997.

[NAB+20] Michael Naehrig, Erdem Alkim, Joppe Bos, Léo Ducas, Karen Easterbrook, Brian LaMacchia, Patrick Longa, Ilya Mironov, Valeria Nikolaenko, Christopher Peikert, Ananth Raghunathan, and Douglas Stebila. FrodoKEM. Technical report, National Institute of Standards and Technology, 2020. available at https://csrc.nist.gov/projects/post-quantum-cryptography/round-3-submissions.

[NJW18] Alexander Nilsson, Thomas Johansson, and Paul Stankovski Wagner. Error amplification in code-based cryptography. *IACR TCHES*, 2019(1):238–258, 2018. https://tches.iacr.org/index.php/TCHES/article/view/7340.

[oSN16] National Institute of Standards and Technology (NIST). Submission requirements and evaluation criteria for the post-quantum cryptography standardization process, 2016. available at: https://csrc.nist.gov/CSRC/media/Projects/Post-Quantum-Cryptography/documents/call-for-proposals-final-dec-2016.pdf.

[Pao10] Gabriele Paoloni. How to Benchmark Code Execution Times on Intel IA-32 and IA-64 Instruction Set Architectures. Technical report, 2010. available at: https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/ia-32-ia-64-benchmark-code-execution-paper.pdf.

[PT19] Thales Bandiera Paiva and Routo Terada. A timing attack on the HQC encryption scheme. In Kenneth G. Paterson and Douglas Stebila, editors, *SAC 2019*, volume 11959 of *LNCS*, pages 551–573. Springer, Heidelberg, August 2019.

[RKL+04] Srivaths Ravi, Paul C. Kocher, Ruby B. Lee, Gary McGraw, and Anand Raghunathan. Security as a new dimension in embedded system design. In Sharad Malik, Limor Fix, and Andrew B. Kahng, editors, *Proceedings of the 41th Design Automation Conference, DAC 2004, San Diego, CA, USA, June 7-11, 2004*, pages 753–760. ACM, 2004.

[Sch21] Leander Schröder. A novel timing side-channel assisted key-recovery attack against HQC. Master's thesis, TU Darmstadt/TU Wien, 2021. https://doi.org/10.34726/hss.2022.91042.

[Sen21] Nicolas Sendrier. Secure sampling of constant-weight words application to bike. Cryptology ePrint Archive, Report 2021/1631, 20211217:142141 (posted 1639750901 17-Dec-2021 14:21:41 UTC), December 2021. https://eprint.iacr.org/2021/1631/20211217:142141.

[SM16] Douglas Stebila and Michele Mosca. Post-quantum key exchange for the internet and the open quantum safe project. In Roberto Avanzi and Howard M. Heys, editors, *SAC 2016*, volume 10532 of *LNCS*, pages 14–37. Springer, Heidelberg, August 2016.

[UXT+22] Rei Ueno, Keita Xagawa, Yutaro Tanaka, Akira Ito, Junko Takahashi, and Naofumi Homma. Curse of re-encryption: A generic power/em analysis on post-quantum kems. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2022(1):296–322, 2022.

[Wel47] Bernard Lewis Welch. The generalisation of student's problems when several different population variances are involved. *Biometrika*, 34(1-2):28–35, 01 1947.

[WSN18] Wen Wang, Jakub Szefer, and Ruben Niederhagen. FPGA-based niederreiter cryptosystem using binary goppa codes. In Tanja Lange and Rainer Steinwandt, editors, *Post-Quantum Cryptography - 9th International Conference, PQCrypto 2018*, pages 77–98. Springer, Heidelberg, 2018.

[WTBB+19] Guillaume Wafo-Tapa, Slim Bettaieb, Loic Bidoux, Philippe Gaborit, and Etienne Marcatel. A practicable timing attack against HQC and its countermeasure. Cryptology ePrint Archive, Report 2019/909, 2019. https://eprint.iacr.org/2019/909.

[YZ17] Yu Yu and Jiang Zhang. Lepton. Technical report, National Institute of Standards and Technology, 2017. available at https://csrc.nist.gov/projects/post-quantum-cryptography/round-1-submissions.