On the Timing Leakage of the Deterministic Re-encryption in HQC KEM

Clemens Hlauschek[1], Norman Lahr[2] 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

Abstract. Well before large-scale quantum computers will be available, traditional cryptosystems must be transitioned to post-quantum 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 of the optimized implementation, for example, with regard to resistance against side-channel attacks.
HQC is a promising code-based key encapsulation scheme and selected as an alternate candidate in the third round of the competition, which puts it on track for getting standardized separately to the finalists, in a fourth round.
Despite having already received heavy scrutiny with regard to side channel attacks, in this paper, we show a novel timing vulnerability in the optimized implementations of HQC, leading to a full secret key recovery. The attack is both practical, requiring only approx. 866,000 idealized decapsulation timing oracle queries in the 128-bit security setting, and structurally different from previously identified attacks on HQC: Previously, exploitable side-channel leakages have been identified in the BCH decoder of a previously submitted version, in the ciphertext check as well as in the PRF of the Fujisaki-Okamoto (FO) transformation employed by several NIST PQC KEM candidates. In contrast, our attack uses the fact that the *rejection sampling routine* invoked during the deterministic re-encryption of the KEM decapsulation leaks secret-dependent timing information. These timing leaks 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. Besides a detailed analysis of the new attack, we discuss possible countermeasures and their limits.

Keywords: Side-channel attack · Rejection sampling · FO transformation · Post-quantum cryptography · HQC

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 exabytes 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, KLM+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 NIST ECC standards [BL15]) emphasize that a good cryptographic design requires simplicity of a secure implementation, and recommend that standardization bodies such as 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 [IIES15]. 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 National Institute of Standards and Technology (NIST) announced a competition [oSN16] which aims to standardize schemes for Post-Quantum Cryptography (PQC) and requests the authors to submit a reference implementation that addresses side-channel attacks in addition to the specification.

Hamming Quasi-Cylic (HQC) [AAB+21] is a promising code-based key encapsulation scheme and an alternate candidate in the third round of the competition. As alternate candidate, HQC might be standardized by NIST in addition to the competition finalists in a fourth round. The Public Key Encryption (PKE) variant of HQC is secure under the Indistinguishability under Chosen Plaintext Attack (IND-CPA) notion. The Key Encapsulation Mechanism (KEM) variant of HQC utilizes the generic quantum-secure Fujisaki-Okamoto (FO) transformation proposed by Hofheinz, Hövelmanns, and Kiltz [HHK17]. It converts the PKE variant to be secure with regard to Indistinguishability under Chosen Ciphertext Attack (IND-CCA). The authors of HQC selected this transformation because it is resistant to the decryption errors which can occur in the HQC decryption procedure. It is also the reason why this transformation is utilized by most NIST PQC lattice-based 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. 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 of the comparison step is to use another constant-time comparison, for example, as provided

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

by OpenSSL[1].

Most recently, Ueno et al. [UXT+21] 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.

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.

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

- an hitherto unconsidered timing variation dependent on the secret key in the deterministic re-encryption of the KEM decapsulation of HQC due to the non-constant time *rejection sampling* function,
- a novel timing attack on the optimized reference implementation of HQC achieving full secret key recovery with high probability, and
- a discussion of possible countermeasure to avoid the identified leakage in the deterministic re-encryption step.

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 in [ABD+16]. 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 $\mathcal{R} = \mathbb{F}_2[X]/(X^n - 1)$.

[1] https://www.openssl.org/docs/man1.1.1/man3/CRYPTO_memcmp.html

Algorithm 1: KeyGen

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

Algorithm 2: Encrypt

```
Input: pk, $\mathbf{m}$, $\theta$
Output: $\mathbf{c} = (\mathbf{u}, \mathbf{v})$
1 $\text{SampleInit}(\theta)$
2 $\mathbf{r_1} = \text{Sample}(\mathcal{R}, \omega_r)$
3 $\mathbf{r_2} = \text{Sample}(\mathcal{R}, \omega_r)$
4 $\mathbf{e} = \text{Sample}(\mathcal{R}, \omega_e)$
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: Decrypt

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

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

2.2 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 $\mathcal{G}$, $\mathcal{K}$, and $\mathcal{H}$. To encapsulate a randomly chosen message $\mathbf{m}$ the randomness θ for the

Algorithm 4: Encaps

```
Input: $\mathsf{pk}$
Output: $K$, $(\mathbf{c}, d)$
1 $\mathbf{m} = \text{Sample}(\mathbb{F}_2)$
2 $\theta = \mathcal{G}(\mathbf{m})$
3 $\mathbf{c} = \text{Encrypt}(\mathsf{pk}, \mathbf{m}; \theta)$
4 $K = \mathcal{K}(\mathbf{m}, \mathbf{c})$
5 $d = \mathcal{H}(\mathbf{m})$
```

Algorithm 5: Decaps

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

encryption is derived by $\mathcal{G}(\mathbf{m})$. The shared key K is a linkage of both the message $\mathbf{m}$ and the ciphertext $\mathbf{c}$ and is computed by $\mathcal{K}(\mathbf{m}, \mathbf{c})$. Finally, d is derived by computing the hash $\mathcal{H}(\mathbf{m})$.

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 $\mathbf{c}'$ matches the received $\mathbf{c}$ and whether the sent digest d equals the hash value of the decrypted message $\mathbf{m}'$. If this check succeeds, $\mathcal{K}(\mathbf{m}, \mathbf{c})$ is output, otherwise failure.

3 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.

3.1 Vulnerability in HQC Implementations

As described in Section 2, the encryption procedure 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 Encrypt function from the message $\mathbf{m}$ by $\mathcal{G}(\mathbf{m})$. 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 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.

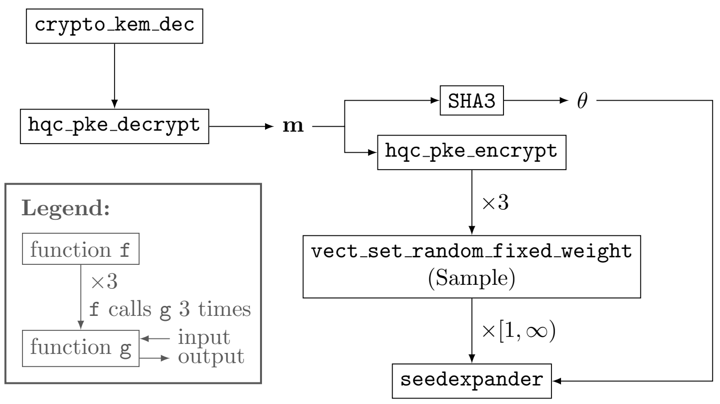


Figure 1: Visualization of the information flow in the decapsulation function of the current HQC KEM implementation [AAB+].

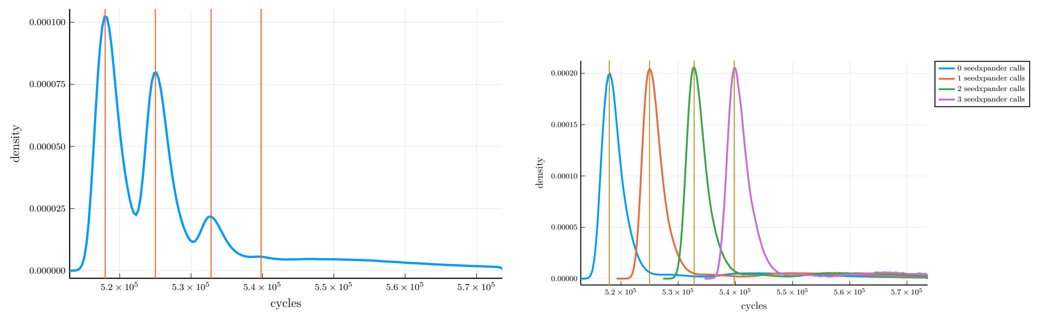
The randomness in the Sample function is deterministic and determined by an eXtendable-Output Function (XOF) implemented by the `seedexpander` function. For our analyses we assume that the outputs of the XOF are uniformly, independent and identically distributed (iid). The XOF influences the path that is taken through the function and is initialized with the seed $\theta = \mathcal{G}(\mathbf{m})$. The message $\mathbf{m}$ is obtained from the decoding of the ciphertext $\mathbf{c}$, 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 $\{0, \ldots, n-1\}$. To generate these positions, the algorithm performs an inner rejection sampling algorithm. The inner rejection sampling algorithm samples a position from $\{0, \ldots, 2^{24}-1\}$ 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 $\{0, \ldots, 2^{24}-1\}$ 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 2: 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



(a) Probability density.

(b) Conditioned probability density.

Figure 2: The probability density and the conditioned probability density, from no to three additional `seedexpander` calls, of the running time of the decapsulation function. The vertical bars are the medians of the runtime of the specific numbers of `seedexpander` calls.

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 2.

Decapsulation timing. To confirm the previously postulated hypothesis on the timing behavior we did a leakage assessment by measuring the CPU cycles on a dedicated machine with no other load of the entire decapsulation function in the hqc-128 setting for ten million random ciphertexts. Fig. 2a shows the corresponding probability density. There are four local maxima, highlighted by the vertical bars which point to the medians of the number of cycles of triggering no additional, but one by default, `seedexpander` call up to three additional `seedexpander` calls. If we condition the probability density to the specific number of `seedexpander` calls, as we show in Fig. 2b, it gets even more clear that the running time of the decapsulation depends on the number of `seedexpander` calls. As expected, the frequency decreases when the number of additional 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 do not emerge in practice.

Inspecting the decapsulation function in Algorithm 5 the timing variation is caused by the invocation of the encryption function using the seed $\theta = \mathcal{G}(\mathbf{m})$. 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`.

3.2 Distinguisher

Given a ciphertext $\mathbf{c}$ 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:

$$\mathcal{D}^{\mathcal{O}}(\mathbf{c}_1, \mathbf{c}_2) = \mathcal{O}(\mathbf{c}_1) \overset{?}{=} \mathcal{O}(\mathbf{c}_2) \tag{1}$$

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 $\cdot \overset{?}{=} \cdot$ returns whether the two arguments are equal or not. The advantage of $\mathcal{D}$ when distinguishing a given ciphertext $\mathbf{c}_1$ that decrypts to $\mathbf{m}_1$ from another ciphertext $\mathbf{c}_2$ that decrypts to a uniform randomly chosen message $\mathbf{m_2}$ is given by:

$$\begin{aligned}
&|\Pr_{\mathbf{c}_2 \leftarrow\!\$\, \mathcal{C}}[\mathcal{D}^{TB(\mathsf{sk},\cdot)}(\mathbf{c}_1, \mathbf{c}_2) = 1 \mid \mathrm{Decrypt}(\mathsf{sk}, \mathbf{c}_1) = \mathrm{Decrypt}(\mathsf{sk}, \mathbf{c}_2)] - \\
&\quad \Pr_{\mathbf{c}_2 \leftarrow\!\$\, \mathcal{C}}[\mathcal{D}^{TB(\mathsf{sk},\cdot)}(\mathbf{c}_1, \mathbf{c}_2) = 1 \mid \mathrm{Decrypt}(\mathsf{sk}, \mathbf{c}_1) \neq \mathrm{Decrypt}(\mathsf{sk}, \mathbf{c}_2)]| \\
= &|\Pr_{\mathbf{c}_2 \leftarrow\!\$\, \mathcal{C}}[TB(\mathsf{sk}, \mathbf{c}_1) = TB(\mathsf{sk}, \mathbf{c}_2) \mid \mathrm{Decrypt}(\mathsf{sk}, \mathbf{c}_1) = \mathrm{Decrypt}(\mathsf{sk}, \mathbf{c}_2)] - \\
&\quad \Pr_{\mathbf{c}_2 \leftarrow\!\$\, \mathcal{C}}[TB(\mathsf{sk}, \mathbf{c}_1) = TB(\mathsf{sk}, \mathbf{c}_2) \mid \mathrm{Decrypt}(\mathsf{sk}, \mathbf{c}_1) \neq \mathrm{Decrypt}(\mathsf{sk}, \mathbf{c}_2)]| \\
= &\ 1 - \Pr_{\mathbf{c}_2 \leftarrow\!\$\, \mathcal{C}}[TB(\mathsf{sk}, \mathbf{c}_1) = TB(\mathsf{sk}, \mathbf{c}_2) \mid \mathrm{Decrypt}(\mathsf{sk}, \mathbf{c}_1) \neq \mathrm{Decrypt}(\mathsf{sk}, \mathbf{c}_2)]
\end{aligned}$$

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 $\mathbf{c}_2$ that decrypts to a different message is at a minimum. We can achieve this by minimizing the probability of the timing behavior of $\mathbf{c}_1$ by picking a suitable message $\mathbf{m}_1$.

3.3 Key Recovery Attack

By using the observations in Section 3.1 to get a distinguisher described in Section 3.2 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 2, 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 $\mathbf{c} = (\mathbf{u}, \mathbf{v})$ 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:

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

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 $\mathbf{c}$ until we detect that the modified ciphertext $\mathbf{c}'$ decrypts to a different message $\mathbf{m}'$. Then, we test for every bit b in the ciphertext $\mathbf{c}'$, 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 $\mathbf{c}'$. 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 $\mathbf{c} = \mathbf{mG} + \mathbf{e}$. (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 $\mathbf{c}$ to obtain $\mathbf{c}'$. If we flip a bit b in the ciphertext $\mathbf{c}'$ and it decrypts back to the original message $\mathbf{m}$, we know that b is an error in $\mathbf{c}' = \mathbf{c} + \mathbf{e} + \mathbf{e}'$. 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 6. First, we need to find a proper message $\mathbf{m}$ which yields 3 additional `seedexpander` calls. Therefor, 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 $\mathbf{c} = (\mathbf{u}, \mathbf{v})$. Further, we define a proper majority threshold N. Afterwards, we apply Algorithm 7 to find another ciphertext $\mathbf{c}' = (\mathbf{u}, \mathbf{y} + \mathbf{e}')$ 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).

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

In particular $\mathbf{c}'$ should have exactly one more error bit than the decoder could correct. From this state, flipping any bit in $\mathbf{c}'$ and checking whether the ciphertext decodes again reveals whether that bit was an error bit in $\mathbf{c}$ or not. We can exploit this property to recover $\mathbf{y}$ later on. Starting from $\mathbf{c}$ 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 $\mathbf{c}'$ 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 6 performs Algorithms 7 and 8 multiple times until a majority is revealed at each bit position for a 0- or 1-bit. To determine the majorities the counters in $\mathbf{t}$ record the total number of votes that have been cast for each bit b. The counters in

Algorithm 6: KeyRecovery

```
Input: Ciphertext $\mathbf{c}$ and majority of $N$.
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      $(\mathbf{c}', \mathsf{bs}) \leftarrow$ FindDiffMsg($\mathbf{c}$)
4      $\mathbf{e}' \leftarrow$ RecoverError($\mathbf{c}'$)
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 $\mathbf{e}$
```

Algorithm 7: FindDiffMsg

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

Algorithm 8: RecoverError

```
Input: Modified ciphertext $\mathbf{c}'$
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 $\mathbf{c}'$
4      if $\mathcal{D}^{\mathrm{TB}(\mathsf{sk},\cdot)}(\mathbf{c}, \mathbf{c}') = 1$ then
5          Set bit $i$ in $\mathbf{e}$
6      end
7      Flip bit $i$ in $\mathbf{v}$ of $\mathbf{c}'$
8  end
```

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 $\lfloor N/2 \rfloor + 1$.

3.4 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 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,

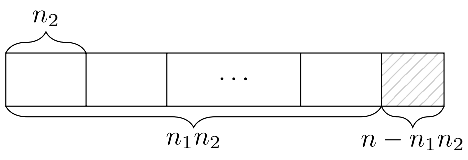


Figure 3: An element of $\mathbb{F}_2[x]/\langle x^n - 1\rangle$ and its segmentation into codewords of the inner code.

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.

3.5 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. 3 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 number of ways to pick these bits is $\sum_{i=0}^{2} \binom{n-n_1 n_2}{i}$. 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 $n - n_1 n_2$ bits. The region o (or "outside") corresponds to the $n_1 n_2$ 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 $Z = Y_{n-n_1 n_2, n_1 n_2, \omega}$. Assuming the attack was successful for all $n_1 n_2$ 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 3. 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.

Table 3: Remaining $n - n_1 n_2$ 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	$n_1 n_2$	n	ω	$n - n_1 n_2$	$\sum_{i=0}^{2} \binom{n-n_1 n_2}{i}$	$\Pr[Z = 0]$	$\Pr[Z \leq 2]$
hqc-128	17,664	17,669	66	5	16	≈ 98.1 %	≈ 100.0 %
hqc-192	35,840	35,851	100	11	67	≈ 97.0 %	≈ 100.0 %
hqc-256	57,600	57,637	131	37	704	≈ 91.9 %	≈ 100.0 %

4 Evaluation

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 [Gab]. 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:

$$\frac{\bar{X}_1 - \bar{X}_2}{\sqrt{\frac{s_{X_1}^2}{N_1} + \frac{s_{X_2}^2}{N_2}}} \tag{4}$$

where $\bar{X}_i$, $s_{X_i}^2$ 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:

$$\nu = \frac{\left(\frac{s_{X_1}^2}{N_1} + \frac{s_{X_2}^2}{N_2}\right)^2}{\frac{\left(\frac{s_{X_1}^2}{N_1}\right)^2}{N_1 - 1} + \frac{\left(\frac{s_{X_2}^2}{N_2}\right)^2}{N_2 - 1}}. \tag{5}$$

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 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 incorrect bits after an attack and show the cumulative distribution function in Fig. 4.

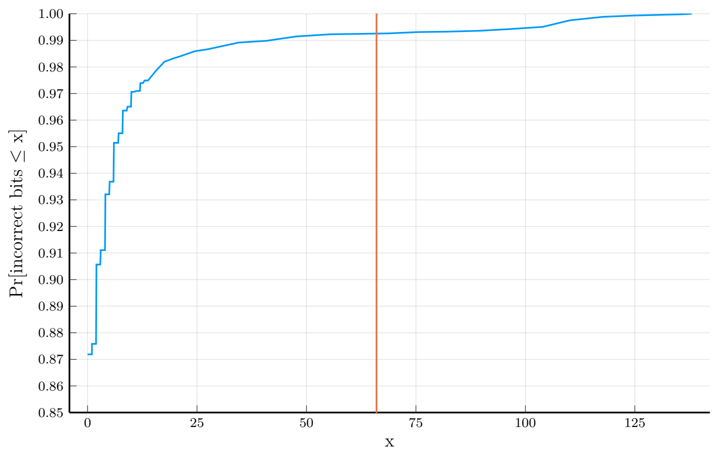


Figure 4: Empirical cumulative distribution function of the number of incorrect bits during the attacks. Approx. 87 % succeeded immediately. For those that failed additional post-processing steps could further improve the success probability. The vertical line indicates the weight of the secret key $\mathbf{y}$.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

5 Discussion on Countermeasures

To counter our proposed attack and to remove the exploitable leakage, we see two basic ways: Either the reencryption step in the decapsulation can be avoided completely or the sampling of a fixed weight bit vector can be implemented in constant time.

Avoiding the reencryption must not harm the Chosen Ciphertext Attack (CCA) security of the KEM. We are not aware of any generic alternative approach that uniquely binds a ciphertext to a message as it is guaranteed by the reencryption step.

We identify two main avenues for implementing a constant-time fixed weight vector sampling algorithm. 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 [Knu, p.145] or reverse sorting, using an established sorting algorithm like merge-sort, as it is proposed in [WSN18] for a ClassicMcEliece hardware implementation. However, the Fisher-Yates shuffling may leak timing information, as it is using secret-dependent array accesses and naïve methods to make these array accesses constant-time result in an unacceptable asymptotic time complexity. The reverse merge-sort induces a slight bias which is solved by a rejection and is therefore not suitable for a constant-time implementation. The Beneŝ-network used in the C reference implementation of ClassicMcEliece is aligned to a vector size of a power of 2 which is not the case in HQC. We are not aware of a suitable random permutation approach for HQC.

Alternatively, we sample the specified number of distinct bit positions in constant-time and set the bits in the vector in constant-time. This method is already implemented for the most part in HQC, except that the distinct position sampling is not constant-time. While we are not aware of an efficient algorithm that implements this method fully correct and in constant-time we propose a way to modify HQC's algorithm to make it 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.

Algorithm 9: 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 $i \bmod m$
```

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 that 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 the following patch to the sampling function:

```
 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};
```

However, the algorithm is not constant-time: rejection sampling still performs a different number of iterations depending on the message. While the countermeasure increases the effort required to perform the attack, it could still allow a local attacker to recover the key.

5.2 Constant-Time Random Number Generation

To further improve our countermeasure we can remove the inner rejection sampling used for generating random indices into the vector. The inner rejection sampling is detailed in Algorithm 9. Instead of rejection sampling integers in the range $0 \le x < \lfloor 2^k/m \rfloor m$, 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 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:

$$\mathrm{SD}_{X,Y} = \frac{1}{2} \cdot \sum_{z \in \Omega} |\Pr[X = z] - \Pr[Y = z]| \tag{6}$$

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

$$\Pr[U_m = z] = \begin{cases} \frac{1}{z} & \text{if } 0 \le z < m \\ 0 & \text{otherwise} \end{cases} \tag{7}$$

Table 4: Statistical distance between the uniform distribution over $\{0, \dots, m-1\}$ 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

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:

$$\Pr[M_n = z] = \begin{cases} \frac{\lfloor n/m \rfloor + 1}{n} & \text{if } 0 \le z < n \bmod m \\ \frac{\lfloor n/m \rfloor}{n} & \text{if } n \bmod m \le z < m \\ 0 & \text{otherwise} \end{cases} \tag{8}$$

The statistical distance between these two distributions is:

$$\mathrm{SD}_{U_m, M_n} = \frac{1}{2} \cdot \sum_{z \in \{0, \dots, m-1\}} \left| \Pr[U_m = z] - \Pr[M_b = z] \right| \tag{9}$$

$$= \frac{1}{2} \cdot \left((n \bmod m) \cdot \left| \frac{1}{m} - \frac{\lfloor \frac{n}{m} \rfloor + 1}{n} \right| + \right. \tag{10}$$

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

In Table 4 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 512 bit non-negative number x modulo a small number efficiently using basic rules of modular arithmetic. We can represent x in base 2^8 as $x = x_0 + 2^8 \cdot x_1 + 2^{8 \cdot 2} \cdot x_2 + \cdots + 2^{8 \cdot (\ell - 1)} x_{\ell - 1} + 2^{8 \cdot \ell} \cdot x_\ell$. We split up the computation of $x \bmod m$ in the following way:

$$x \bmod m = \left(\cdots \overbrace{\left(x_{\ell-1} + 2^8 \cdot \underbrace{(x_\ell \bmod m)}_{z_0} \right) \bmod m}^{z_1} \cdots \right) \bmod m \tag{11}$$

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

$$z_i = \begin{cases} x_\ell \bmod m & \text{if } i = 0 \\ (x_{\ell - i} + 2^8 \cdot z_{i-1}) \bmod m & \text{otherwise} \end{cases} \tag{12}$$

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` by the following lines of C:

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

```c
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;
}
```

where GEN_BYTES is $\frac{b}{8}$.

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 [MVOV, p.603]. This can be observed in Fig. 5. 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[2]. 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.

5.3 Performance Optimization

We wish to minimize the number of random bytes generated, while still ensuring that we only have to call the `seedexpander` function once, and never inside the for-loop of the Sample 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:

$$\Pr[X_{n,i,p_s} = w] = \begin{cases} 0 & \text{if } i < w \\ 1 & \text{if } w = i = 0 \\ p_s & \text{if } w = i = 1 \\ \begin{array}{r} p_s \dfrac{w}{n} \Pr[X_{n,i-1,p_s} = w] + \\ (1 - p_s \max(0, \dfrac{w-1}{n})) \Pr[X_{n,i-1,p_s} = w-1] \end{array} & \text{otherwise} \end{cases} \tag{13}$$

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

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

Table 5: 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}	$\log_2(1-(\Pr[U_{n,\omega_r,p_s} \leq \kappa_{p_s}])^3)$	κ_1	$\log_2(1-(\Pr[U_{n,\omega_r,1} \leq \kappa_1])^3)$
hqc-128	99	≈ -134	97	≈ -129
hqc-192	152	≈ -193	146	≈ -195
hqc-256	192	≈ -261	190	≈ -259

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:

$$\Pr[X_{n,i,p_s} \geq w] = \sum_{x=w}^{i} \Pr[X_{n,i,p_s} = x] \tag{14}$$

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:

$$\Pr[U_{n,w,p_s} \leq i] = \Pr[X_{n,i,p_s} \geq w] \tag{15}$$

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:

$$1 - (\Pr[U_{n,\omega_r,p_s} \leq \kappa])^3. \tag{16}$$

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 5. 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 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 RNG by using the full width of the 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 5 for this purpose. Using these parameters the probability that we sample a vector of too low weight is $\leq 2^{-\lambda}$, 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 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*.

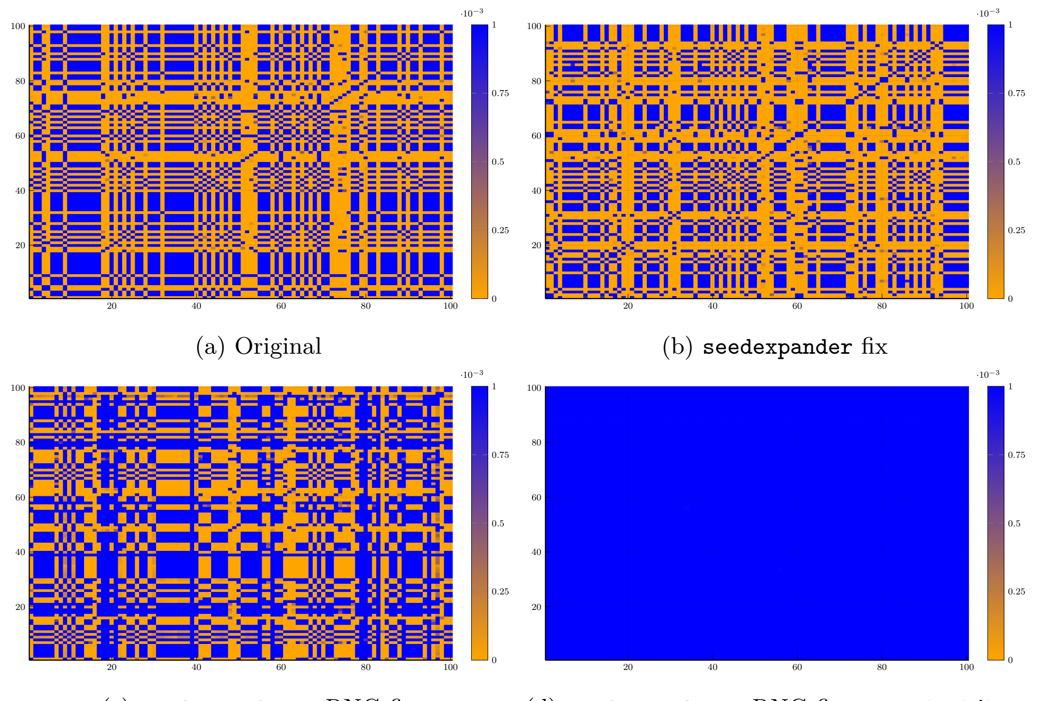
The next phase of the algorithm uses Advanced Vector Extensions (AVX2) instructions to set the sampled bit positions in the bit-array. This algorithm is vectorized to process the bit-array in 256 bit chunks. We modify this algorithm to only include a position if `take[i]` is set by computing a bit mask 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 $0 \oplus x = x$.

```
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.



(a) Original (b) `seedexpander` fix

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

Figure 6: 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.

5.5 Evaluation of the Proposed Countermeasures

The results can be viewed in Fig. 6. 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. 6d. 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 6. 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

Table 6: 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%)

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.

6 Conclusions, Lessons, and Future Work

We have presented a novel timing attack on the optimized implementation of HQC's KEM. The considered implementation in this work has been found vulnerable despite the claim of the authors of HQC "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].

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, non-constant time rejection sampling becomes a problem. This highlights the problem of providing high level definitions of a cryptosystem: The definition is good enough for an implementor 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 [AAB+21]. Therefore, the flaw we identify would likely be implemented by any implementor. This problem also highlights the need for automated, possibly standardized tools to check implementations for secret-dependent timing variations.

Our proposed countermeasure does incur a heavy performance degradation. However, it does eliminate all timing-variation that we could detect from the analyzed function. Future work could focus on improving the performance of said countermeasure through the use of SIMD instructions or different algorithms.

Our attack completely breaks the security guarantees that HQC attempts to uphold. Designers of other schemes should ensure that their cryptosystem does not make use of rejection sampling seeded with values that depend on secrets. In general it is advisable to verify that any values that are derived from secrets such as the secret key or the message do not cause a timing variation in any of the schemes' algorithms. We have identified that a timing variation very similar to the one discovered in HQC exists in BIKE [ABB+20]. However, it is unclear whether the timing variation in BIKE is also exploitable, especially given the constraint that BIKE considers timing attacks out of scope as keys are not meant to be reused [ABB+20].

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. 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, Phillipe 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.

[ABD+16] Carlos Aguilar, Olivier Blazy, Jean-Christophe Deneuville, Philippe Gaborit, and Gilles Zémor. Efficient encryption from random quasi-cyclic codes. Cryptology ePrint Archive, Report 2016/1194, 2016. https://eprint.iacr.org/2016/1194.

[AP13] N. J. Al Fardan and Kenneth G. Paterson. Lucky Thirteen: Breaking the TLS and DTLS Record Protocols. In *2013 IEEE Symposium on Security and Privacy*, pages 526–540. IEEE, may 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, *Advances in Cryptology – EUROCRYPT 2019, Part II*, volume 11477 of *Lecture Notes in Computer Science*, pages 747–776, Darmstadt, Germany, May 19–23, 2019. Springer, Heidelberg, Germany.

[BDL97] Dan Boneh, Richard A. DeMillo, and Richard J. Lipton. On the Importance of Checking Cryptographic Protocols for Faults. In *EUROCRYPT 1997: Advances in Cryptology — EUROCRYPT '97*, volume 1233, pages 37–51. 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, mar 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.

[BL15] Daniel J Bernstein and Tanja Lange. Failures in NIST's ECC standards. pages 1–27, 2015.

[BT11] Billy Bob Brumley and Nicola Tuveri. Remote Timing Attacks Are Still Practical. In *ESORICS 2011: Computer Security – ESORICS 2011*, volume 6879 LNCS, pages 355–371. 2011.

[CWR09] Scott A. Crosby, Dan S. Wallach, and Rudolf H. Riedi. Opportunities and Limits of Remote Timing Attacks. *ACM Transactions on Information and System Security*, 12(3):1–29, jan 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. *Proceedings of the ACM SIGCOMM Internet Measurement Conference, IMC*, pages 475–488, 2014.

[Dya18] Mikhail Dyakonov. The case against quantum computing. IEEE Spectrum, 2018.

[Gab] Gabriele Paoloni. How to Benchmark Code Execution Times on Intel® IA-32 and IA-64 Instruction Set Architectures.

[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, *Advances in Cryptology – CRYPTO 2020, Part II*, volume 12171 of *Lecture Notes in Computer Science*, pages 359–386, Santa Barbara, CA, USA, August 17–21, 2020. Springer, Heidelberg, Germany.

[HGS99] Chris Hall, Ian Goldberg, and Bruce Schneier. Reaction attacks against several public-key cryptosystems. In Vijay Varadharajan and Yi Mu, editors, *ICICS 99: 2nd International Conference on Information and Communication Security*, volume 1726 of *Lecture Notes in Computer Science*, pages 2–12, Sydney, Australia, November 9–11, 1999. Springer, Heidelberg, Germany.

[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: 15th Theory of Cryptography Conference, Part I*, volume 10677 of *Lecture Notes in Computer Science*, pages 341–371, Baltimore, MD, USA, November 12–15, 2017. Springer, Heidelberg, Germany.

[Hog15] Mél Hogan. Data flows and water woes: The Utah Data Center. *Big Data & Society*, 2(2):205395171559242, dec 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.

[IIES15] Gorka Irazoqui, Mehmet Sinan Inci, Thomas Eisenbarth, and Berk Sunar. Lucky 13 strikes back. *ASIACCS 2015 - Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security*, pages 85–96, 2015.

[Kal20] Gil Kalai. The Argument against Quantum Computers, the Quantum Laws of Nature, and Google's Supremacy Claims. pages 1–33, 2020.

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

[KLM+04] Paul Kocher, Ruby Lee, Gary McGraw, Anand Raghunathan, and Srivaths Ravi. Security as a new dimension in embedded system design. *Proceedings - Design Automation Conference*, pages 753–760, 2004.

[Knu] Donald E Knuth. *The Art of Computer Programming.* Addison-Wesley.

[Koc96] Paul C Kocher. Timing Attacks on Implementations of Diffie-Hellman. *CRYPTO - Annual International Cryptology Conference*, pages 104–113, 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 *CANS 2016: Cryptology and Network Security*, volume 10052 LNCS, 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.

[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). *Proceedings of the 30th USENIX Security Symposium*, pages 213–230, 2021.

[Mos17] Michele Mosca. The quantum threat to cybersecurity (for cxos). Presented at the 5th ETSI-IQC Workshop on Quantum-Safe Cryptography, 2017.

[MSEH20] Daniel Moghimi, Berk Sunar, Thomas Eisenbarth, and Nadia Heninger. TPM-FAIL: TPM meets timing and lattice attacks. *Proceedings of the 29th USENIX Security Symposium*, pages 2057–2073, 2020.

[MVOV] A. J. Menezes, Paul C. Van Oorschot, and Scott A. Vanstone. *Handbook of Applied Cryptography.* CRC Press Series on Discrete Mathematics and Its Applications. CRC Press.

[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.

[oSN16] National Institute of Standards and Technology (NIST). Submission requirements and evaluation criteria for the post-quantum cryptography standardization process, 2016. https://csrc.nist.gov/CSRC/media/Projects/Post-Quantum-Cryptography/documents/call-for-proposals-final-dec-2016.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: 26th Annual International Workshop on Selected Areas in Cryptography*, volume 11959 of *Lecture Notes in Computer Science*, pages 551–573, Waterloo, ON, Canada, August 12–16, 2019. Springer, Heidelberg, Germany.

[UXT+21] 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. In *IACR-TCHES-2022*, pages 1–26, 2021.

[WEL47] B. L. WELCH. THE GENERALIZATION OF 'STUDENT'S' PROBLEM WHEN SEVERAL DIFFERENT POPULATION VARLANCES 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, Fort Lauderdale, Florida, United States, April 9–11 2018. Springer, Heidelberg, Germany.

[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.