TU WIEN Informatics | TECHNISCHE UNIVERSITÄT DARMSTADT

A Novel Timing Side-Channel Assisted Key-Recovery Attack Against HQC

MASTER'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Master of Science

in

IT Security

by

Robin Leander Schröder, BSc
Registration Number 01427251

to the Faculty of Informatics

at the TU Wien

Advisor:	Univ. Prof. DI Dr. Thomas Grechenig
Assistance:	Clemens Hlauschek

at the TU Darmstadt

Advisor:	Prof. Dr. Michael Waidner
Assistance:	Norman Lahr

1st August, 2021

Robin Leander Schröder	Michael Waidner	Thomas Grechenig

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

Erklärung zur Verfassung der Arbeit

Robin Leander Schröder, BSc

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Erklärung zur Abschlussarbeit gemäß § 22 Abs. 7 und § 23 Abs. 7 APB TU Darmstadt

Hiermit versichere ich, Robin Leander Schröder, die vorliegende Master-Thesis / Bachelor-Thesis gemäß § 22 Abs. 7 APB der TU Darmstadt ohne Hilfe Dritter und nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die Quellen entnommen wurden, sind als solche kenntlich gemacht worden. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen. Mir ist bekannt, dass im Falle eines Plagiats (§38 Abs.2 APB) ein Täuschungsversuch vorliegt, der dazu führt, dass die Arbeit mit 5,0 bewertet und damit ein Prüfungsversuch verbraucht wird. Abschlussarbeiten dürfen nur einmal wiederholt werden. Bei der abgegebenen Thesis stimmen die schriftliche und die zur Archivierung eingereichte elektronische Fassung gemäß § 23 Abs. 7 APB überein.

1. August 2021

Robin Leander Schröder

Abstract

Recently, multiple timing side-channels have been found in the Hamming Quasi-Cyclic (HQC) implementation. These enable three attacks that enable an attacker to recover the secret key from timing leakage. Two of them exploit an issue with the use of a non-constant time BCH decoder. The remaining one stems from using a non-constant time comparison.

We demonstrate that another critical timing side-channel vulnerability exists in the HQC implementation. The timing side-channel is rooted in the use of values derived from secrets as a seed in a rejection sampling procedure. We forge multiple attacks exploiting the timing side-channel. We evaluate and improve their complexity and success probability. The fastest attack requires only 19,942 decapsulation oracle calls and succeeds with an empirically determined probability of $\approx 96.7\%$.

Our research demonstrates that timing attacks are long from gone – in post-quantum cryptography they can shatter the security of a system even in the face of a classical attacker. Luckily, the issues identified in this thesis are not a death-sentence for the cryptosystem, and can be remedied using the countermeasures we proposed. While our main countermeasure has a heavy impact on total decapsulation time (+41%) we demonstrate that we cannot detect any statistically significant residual timing-variation in the patched version.

Keywords: HQC, timing side-channel attack, post-quantum cryptography

Contents

CHAPTER 1

Introduction

The world has enjoyed the efficiency of classical algorithms that have undergone a long period of optimization. However, with the rise of quantum computing a new threat rises on the horizon. Due to Shor's algorithm [Sho94; DMR11] most asymmetric cryptographic primitives in use today may be broken using a sufficiently large scale quantum computer. While it is not known whether it is feasible to build a quantum computer large enough to break the cryptography we use today, it may be a worthwhile endeavour to secure our systems before a sufficiently powerful quantum computer surprises us. In fact, we would like to secure these systems long before large scale quantum computers become a reality, as adversaries could record transmissions secured by a key-exchange and store the transmission until the required quantum computer becomes available. To protect against such an adversary we should protect against a potential future quantum adversary using quantum resilient cryptography starting today.

The importance of this effort is underlined by the National Institute of Standards and Technology (NIST) which have set out to standardize post-quantum Key Encapsulation Mechanisms (KEMs) (e.g., *CRYSTALS-KYBER* [Sch+20] or *SABER* [D'A+20]), and signature schemes (e.g., *CRYSTALS-DILITHIUM* [Lyu+20] or *Rainbow* [Din+20]). Such quantum resistant or "post-quantum" cryptography has been in development for a while. In particular, code-based cryptography, first published by McEliece in 1978 [Rob78], has found new interest, as it is conjectured to be resistant to quantum empowered attacks. It bases its security properties on the hardness of decoding certain kinds of Error Correcting Codes (ECCs).

There are several methods in the state of the art by which we attempt to forge post-quantum cryptography. Next to code-based cryptography the arguably most prominent schemes base their hardness on problems in lattices (such as the aforementioned candidates: *CRYSTALS-KYBER* [Sch+20] or *SABER* [D'A+20]). There have been several attempts at using hard problems in multivariate polynomials to create KEMs. The two NIST proposals were *CFPKM* [CFP17] and *Giophantus* [Aki+17]. However, neither of them

survived the first NIST standardization round [CSD17b]. Hash-based signatures (e.g., *SPHINCS+* [Hul+20]) are an interesting contender, as they rely only on the hardness of breaking the underlying hash-function.

Unfortunately, post-quantum KEMs do not yield the same performance characteristics as current pre-quantum schemes [Mea]. For example, a Elliptic-Curve Diffie-Hellman (ECDH) based KEM [KL14, p.405] instantiated with the x25519 Diffie-Hellman (DH) function [Ber06; Dan14] sports key and ciphertext sizes of 32 bytes [Jed]. At the time of writing, post-quantum KEMs with the smallest keys are based on supersingular isogenies [Mea]. The smallest instantiation of these schemes offers a public-key size of 330 bytes and a ciphertext size of 346 bytes [Jao+20]. However, the computation required for these schemes is an order of magnitude larger than many lattice or code-based schemes [Mea]. Additionally, supersingular isogenies are still a relatively fresh contender in the cryptographic community: the first schemes basing their security on the hardness of traversing the supersingular isogeny graph were proposed in 2006 [CGL06; Eis+18]. While the research on elliptic curves can partially be transferred to supersingular isogenies, there has been comparatively little cryptanalysis for these schemes.

Besides supersingular isogenies, the leading candidates in terms of ciphertext-size and performance are lattice-based schemes, although code-based schemes follow shortly behind [Mea], and may under certain circumstances be the more performant choice: some of the algorithms in code-based cryptography can be more performant than algorithms in lattice-based when implemented on Field-Programmable Gate Arrays (FPGAs) [Agu+19; D'A+19]. Due to these circumstances, code-based cryptography offers a compelling tradeoff in security, performance and size.

Timing attacks pose an imminent threat to hardware and software implementations of cryptography [Han]. Contrary to many other side-channel attacks timing side-channels are often exploitable remotely, and do not require an adversary to have physical access to the hardware that a cryptographic primitive is run on. Exploitation of such a vulnerability can lead to a full compromise of keys, plaintexts or other secrets. A breakage at this low level erodes the very foundation that modern systems build their security guarantees upon. In the worst case, all an attacker needs is network access to a server that responds to encrypted requests. Given that much of today's world is connected through the internet these prerequisites are easily fulfilled for many potential targets.

Attacks exploiting timing side-channels have persisted for almost three decades [Koc96] and crept up again and again even in well reviewed code [AFP13; AP16; Ber05]. Many implementors of cryptographic primitives now take great care to achieve constant time implementations. The NIST post-quantum evaluation-criteria acknowledge this and state that optimized implementations bear more weight when they are implemented in a constant time fashion [CSD17c].

HQC [Mel+20] is a code-based post-quantum KEM and has advanced as an alternative candidate to the latest round 3 in the NIST effort for standardization of post-quantum cryptography [CSD17a]. In contrast to, for example, the Classic McEliece [Ber+19]

submission, HQC offers reduced public-key sizes.

Recently, a plethora of timing side-channels have been identified in HQC [Hqcd; GJN20; WT+19; PT19]. This elicits the question, whether any timing side-channels remain, or whether the scheme has received sufficient expert review. The so far perpetual reemergence of timing-side channels in code used by millions seems to suggest the former.

1.1 Research Goals

During preliminary analysis for the thesis we have identified a timing side-channel enabling key recovery in HQC. The goal of the thesis is fivefold:

1. Explain the attack and its discovery.
2. Optimize the discovered attack.
3. Relate the attack to existing attacks.
4. Determine which KEMs are affected.
5. Investigate potential countermeasures.

1. Attack and Discovery Finding the timing side-channel and an attack that can exploit its leakage is non-trivial. We explain the methodology used to discover the side-channel in detail and contrast it to other approaches. The attack is explained and its runtime and success probability analyzed. If we find better attack strategies, we show how the attack evolved.

2. Attack Optimization The attack we discovered requires on the order of 3 to 11 million calls to an idealized decapsulation oracle, which yields perfect timing information of the underlying timing variation. To measure the side-channel in the real world, one must perform approximately 10,000 decryption calls for each idealized decryption call. Therefore, in practice the attack requires on the order of $2^{34.8}$ to $2^{36.7}$ decapsulations. This is feasible, however far from ideal.

To improve the information rate extracted from the side-channel, the attack strategy must be improved. There are various ways this could be achieved, and some may require analyzing the structure of the Reed-Muller (RM) and Reed-Solomon (RS) codes used by HQC.

Timeless attacks [van+20] offer a way to drastically reduce the number of timing measurements that must be performed. Even if we cannot find an improvement for the attack strategy itself, this attack methodology can cut the required number of real world decryption calls by up to a factor of 10,000.

Ideally, both methods succeed and the final attack will require a tremendously lower number of decapsulations.

3. Attack Classification The attack strategy is similar to the one outlined in [GJN20]. However, this paper only details the attack strategy for lattice-based schemes. As part of literature review we will determine the novelty of the attack.

4. Attack Impact Preliminary analysis determined, that the side-channel is not exclusive to HQC: Bit Flipping Key Encapsulation (BIKE) [Ara+19], another code-based KEM, also seems to contain a similar issue. However, it is unknown whether the timing variation is exploitable. Even though the violating part of the implementation is the same, the information gained by the timing variations are different, and will likely lead to a completely different attack strategy, should the side-channel be exploitable.

To understand how general the found timing side-channel is, we must look at other schemes and determine whether they are also impacted. Similar to a recent attack [GJN20] on schemes constructed using the Fujisaki Okamoto (FO) transform [HHK17], a larger number of schemes may be impacted.

5. Countermeasures Removing the side-channel is non-trivial, especially when we wish to retain as much performance as possible. Therefore, we investigate how a constant-time algorithm solving the same problem may be constructed. Tapping into the rich fountain of prior research will likely aid in the search.

1.2 Methodology

Literature review of the large available corpus will be one of the main tools, to familiarize oneself with the state of the art in coding-theory (for example [HP03; Rob06]), code-based cryptography (for example [Dra+18; Lö15; Meu12; Agu+16]), timing attacks (for example [WT+19; Eat+18; GJN20; PT19]) and other side-channel attacks on post-quantum schemes (for example [TW19; Bru+16]).

In some instances, methods from reaction attacks that are rooted in a high Decryption/Decoding Failure Rate (DFR) have later been used in timing-attacks. Therefore, the class of reaction attacks is imperative to understand [Fab+17; CMM19; DRV19].

Implementation HQC is implemented in C. Therefore, it is natural to implement at least parts of the attack in C as well. Data analysis from empirical measurements is performed using Julia[1], which offers excellent facilities for statistical tests or plotting. Other experiments, for example, to test new attack strategies on simplified instances, are performed using sagemath [Ste+20].

Finding Timing Leaks We determine the existence of a timing variation by measuring the number of cycles the decapsulation algorithm takes on different ciphertexts. Welch's t-test then determines whether the timing difference is statistically significant.

[1] https://julialang.org/

Measuring the number of cycles a computation takes can be done using the `rdtscp` instruction on a modern x86 ISA. Several microarchitectural elements on modern CPUs complicate taking accurate measurements. One such factor is the cache, which can be directly controlled using the `clflush` instruction. However, other factors such as the branch prediction unit and its buffers, or the prefetcher are harder to account for and may introduce unwanted biases in the measurements. However, if the timing variation is strong enough, these factors may vanish in comparison to the variations that we look for.

1.3 Structure of this Thesis

We divide this thesis into 5 main parts. The first one of which you have now almost completed is the introduction. Following the introduction we will survey the foundations required for understanding HQC and timing attacks. A major share will be dedicated to understanding the fundamentals of coding theory in Section 2.2. Here, we introduce the codes that are used in HQC and the problems that HQC reduces its security to. We then give a brief introduction to side-channel attacks and in particular timing-attacks in Section 2.4. In Chapter 3 we introduce the HQC scheme and briefly present prior timing attacks on the scheme. The main contribution of this thesis is in Chapter 4. Here, we illustrate how a timing-variation in the HQC implementation is fatal to the security of the cryptosystem. We explain the discovery of the timing-variation in detail. To prove the exploitability of the timing-variation we present two new attacks which recover the secret key. We derive estimates the success probability of each attack and empirically evaluate these attacks. We establish several countermeasures and assess their strengths and weaknesses. We thoroughly evaluate the most practical countermeasure and demonstrate that while it does result in a significant performance impact, it eradicates all timing-variations which we could detect. Then, we show that a structurally similar timing-variation exists in the BIKE KEM in Chapter 8. Finally we conclude our research in Chapter 9.

CHAPTER 2

Fundamentals

In this chapter we give a brief introduction to coding-theoretic foundations and problems which code-based cryptography can base its security on. We then introduce some of the earliest code-based cryptosystems. Furthermore, we introduce timing-attacks and show several methods to identify timing-variations. Finally, we discuss how implementations can mount a defense against these attacks by implementing countermeasures.

2.1 Notation

We use the following notation:

$[\![m]\!]$	The set $\{1, \ldots, m\}$ containing the integers from 1 to m (inclusive).
$a_T = Y$	a is equal to Y on the coordinates in T.
a_T	may also be viewed as a projection: only the coordinates in T from a are selected.
$\mathbf{1}^n$	a vector of ones with length n.
$x \leftarrow\!\!\$\, S$	x is sampled randomly from S using some distribution; typically the uniform distribution.
$\bar{S}$	the set complement of the set S defined by $\{1, \ldots, m\} \setminus S$ for some m
$\mathbb{F}_q$	the finite field of order q. For example, $\mathbb{F}_2$ is the binary field containing the elements 0 and 1 – the additive and multiplicative identity.
$\mathbf{s}, \mathbf{x}, \mathbf{y}$	Vectors are identified by lowercase letters in bold font.
$\mathbf{H}, \mathbf{G}$	Matrices are identified by uppercase letters in bold font.
$\langle \mathbf{x}, \mathbf{y} \rangle$	is the inner product of $\mathbf{x}$ and $\mathbf{y}$: $\langle \mathbf{x}, \mathbf{y} \rangle = \sum_i \mathbf{x}_i \mathbf{y}_i$.
$\langle x^n - 1 \rangle$	The ideal generated by the element $x^n - 1$.
$(v_1, v_2, \ldots, v_n)$	A row vector of length n containing the elements $(v_i)_{1 \le i \le n}$.

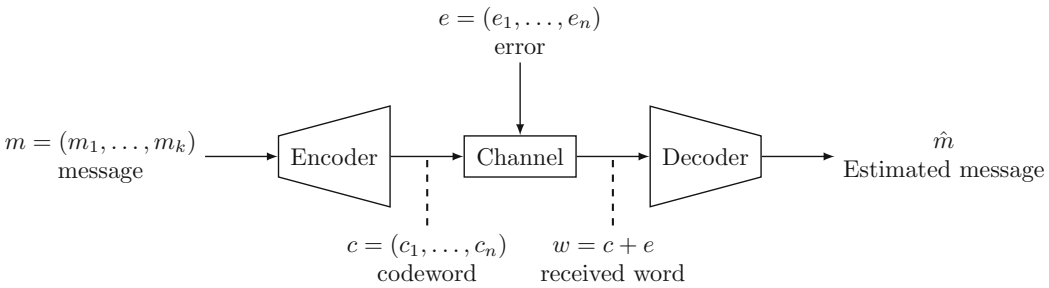


Figure 2.1: Communication model in coding-theory: a message is encoded, sent across a noisy channel and decoded at the receiver. The codeword must be longer than the message for the decoder to correct errors induced by the channel (c.f. [HP03, p.2]).

2.2 Coding-Theory

Error Correcting Codes and coding theory are concerned with transmitting information across a noisy channel that induces errors [HP03; Rob06]. A simple solution to this problem would be to add a checksum such as a CRC or a parity bit to detect errors. Then, the receiver could request retransmission upon detecting an error. However, this becomes infeasible when the sender of the message is not immediately available at the point in time or space that the message is read. This may be the case in, for example, space missions where there is a large latency due to the spacial separation, or when storing data on physical media such as RAM or flash memory which is to be read back at a later point in time. Additionally, if the channel is very noisy, only very few message may transmit completely error-free.

To remediate this situation we can use coding theory as depicted in Fig. 2.1: the sender encodes the message by means of an encoding algorithm which adds redundancy to the message that allows the receiver to recover the original message by means of a decoding algorithm even in the face of errors. If there are not too many errors the codeword is decoded successfully and the encoded message $\mathbf{m}$ equals the decoded message $\hat{\mathbf{m}}$. In general, any code $\mathcal{C}$ encodes messages $\mathbf{m} \in \mathbb{F}_q^k$ from an alphabet $\mathbb{F}_q$ of length k. The encoding algorithm results in a codeword $\mathbf{c} \in \mathbb{F}_q^n$ of length $n \geq k$. One of the fundamental properties of a code is its minimum distance.

Definition 2.2.1 (Minimum distance)**.** The *minimum distance* $d_{\min}$ of a code $\mathcal{C}$ with respect to a metric d is the minimum distance between any two codewords:

$$d_{\min} = \min_{\mathbf{c},\mathbf{c}' \in \mathcal{C}, \mathbf{c} \neq c'} d(\mathbf{c}, \mathbf{c}')$$

In this thesis we only consider the Hamming-metric, however many other metrics exist and attempts have been made to construct code-based cryptography using these metrics [Agu+16].

Definition 2.2.2 (Hamming metric)**.** The Hamming metric counts the number of coordinates in which two vectors differ:

$$d(\mathbf{c}, \mathbf{c}') = |\{i \in [\![n]\!] \mid \mathbf{c}_i \neq \mathbf{c}'_i\}|.$$

The minimum distance determines the number of errors that a code may correct. Given a minimum distance d a code $\mathcal{C}$ can correct up to $t = \left\lfloor \frac{d-1}{2} \right\rfloor$ errors when decoding to the nearest codeword. This is also illustrated in Fig. 2.2. Later we will introduce codes that allow us to give good lower-bounds for their minimum distance.

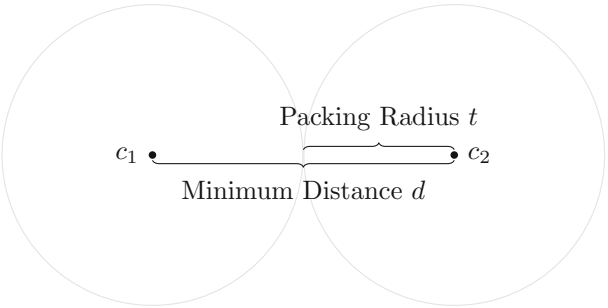


Figure 2.2: At least one pair of codewords c_1 and c_2 has a distance that is the minimum distance d. The corresponding packing radius is $t = \left\lfloor \frac{d-1}{2} \right\rfloor$. Under the assumption that the closest codeword to a received word is the codeword to decode to we cannot correct more than t errors.

Definition 2.2.3 (Rate)**.** The rate of a code $\mathcal{C}$ with parameters $[n, k]$ is: $\frac{k}{n}$

The optimal rate would be 1, however then we could also not introduce any redundancy to the codewords. Thus, coding theory is concerned with finding high rate codes for a given minimum distance. Alternatively, for a given n, k and finite field of order q we would like to find the maximum possible minimum distance d [HP03, p.48].

2.2.1 Linear Codes

A linear code $\mathcal{C}$ is a linear k dimensional subspace of $\mathbb{F}_q^n$. The encoding is a linear map $\mathbf{G}: \mathbb{F}_q^k \to \mathbb{F}_q^n$ which generates the code. $\mathbf{G}$ may be represented as a $k \times n$ matrix, if the messages are row vectors of length k, and is a basis for the code:

$$\mathcal{C} = \left\{ \mathbf{mG} \mid \mathbf{m} \in \mathbb{F}_q^k \right\} \tag{2.1}$$

$\mathbf{G}$ is also called the generator matrix of $\mathcal{C}$. We can also represent a linear code by its parity check matrix $\mathbf{H}$.

Definition 2.2.4 (Parity-Check Matrix)**.** The parity-check matrix $\mathbf{H} \in \mathbb{F}_q^{(n-k)\times n}$ of a $[n, k]$ code $\mathcal{C}$ maps codewords in $\mathcal{C}$ to the zero-vector:

$$\mathbf{H}\mathbf{x}^\intercal = \mathbf{0}^{n-k} \tag{2.2}$$

The parity check matrix is non-empty (i.e. has a non-zero number of rows and columns), when $n > k$: since $\mathcal{C}$ is a proper subspace of $\mathbb{F}_q^n$, there is a linear transformation which has $\mathcal{C}$ as its kernel [HP03, p.4].

Definition 2.2.5 (Syndrome)**.** The syndrome $\mathbf{y}$ of a word $\mathbf{w} \in \mathbb{F}_q^n$ in a linear code $\mathcal{C}$ is given by:

$$\mathbf{y} = \mathbf{H}\mathbf{w}^\intercal \tag{2.3}$$

The code $\mathcal{C}$ consists of all words with a syndrome of $\mathbf{0}_{n-k}$. The parity check matrix also generates a code, called the dual or orthogonal of $\mathcal{C}$. It is denoted by $\mathcal{C}^\perp$, and contains the words that are orthogonal to $\mathcal{C}$. The orthogonality follows from the definition of $\mathbf{H}$ and $\mathcal{C}$.

Definition 2.2.6. The dual code $\mathcal{C}^\perp$ contains the words that are orthogonal to codewords of $\mathcal{C}$:

$$\mathcal{C}^\perp = \left\{ \mathbf{x} \in \mathbb{F}_q^n \mid \forall \mathbf{c} \in \mathcal{C}.\, \langle \mathbf{x}, \mathbf{c} \rangle = 0 \right\} \tag{2.4}$$

Repetition Code

One of the simplest codes is the repetition code. For a given multiplicity μ, the repetition code repeats the message μ times. For example, the generator matrix of a repetition code with $\mu = 3$ is:

$$\mathbf{G}_{R_{1,3}} := \begin{bmatrix} 1 & 1 & 1 \end{bmatrix}.$$

The message (0) is encoded to $(0, 0, 0)$ and the message (1) to $(1, 1, 1)$. We can further extend the repetition code to accept multiple elements of the alphabet:

$$\mathbf{G}_{R_{2,3}} := \begin{bmatrix} 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 \end{bmatrix}.$$

The message $(0, 1)$ is encoded to $(0, 0, 0, 1, 1, 1)$ by this generator matrix. Decoding these codes is simple: for each block of μ bits we decode that block to the majority of these bits, if there is one. Otherwise, the decoding fails, but can signal an error. Unfortunately, these codes have a poor rate: $\frac{1}{\mu}$.

Lemma 2.2.1 (Minimum distance for linear codes)**.** For linear codes the minimum distance between any two codewords is equivalent to the minimum non-zero weight of any codeword [Lö15, p.39]:

$$\begin{aligned} d_{\min} &= \min_{\mathbf{c},\mathbf{c}',\mathbf{c}\neq\mathbf{c}'\in\mathcal{C}} d(\mathbf{c},\mathbf{c}') \\ &= \min_{\mathbf{c},\mathbf{c}',\mathbf{c}\neq\mathbf{c}'\in\mathcal{C}} w(\mathbf{c}-\mathbf{c}') \\ &= \min_{\mathbf{c}\in\mathcal{C},\mathbf{c}\neq 0} w(\mathbf{c}) && \text{By } \mathbf{c}-\mathbf{c}'\in\mathcal{C} \end{aligned}$$

In the following sections we will introduce a range of codes that are still frequently in use today and achieve much higher rates than the repetition code and are also useful for code-based cryptography.

2.2.2 Channel Error Models

We can model the channel across which the information is sent in several ways. The channel model describes the distribution of errors. The binary symmetric channel and the fixed-error model are most important for our discussion of code-based cryptography.

Binary Symmetric Channel

In the binary symmetric channel BSC_ρ, errors are elements of $\{0,1\}$ and a 1 has a fixed probability ρ of occurring. The probability of an error occurring is independent of both the message and other errors, and can thus be modeled by independent Bernoulli variables [Lö15, p.41]. This model is used in HQC to bound the DFR [Car+20, p.18].

Fixed-Error Model

In the fixed-error model we assume the error to have a fixed weight. This model is used in McEliece, where an error of a fixed weight is added to a codeword [Rob78].

2.2.3 Reed-Muller Codes

RM codes are a family of linear codes parameterized by r and m [HP03; Rob06]. The message space of an r^{th} order RM code $R(1,m)$ are the multivariate polynomials with binary coefficients in m variables of degree up to r with the degree of each variable x_i no larger than 1. There are $\binom{m}{0} = 1$ such monomials of degree 0 (namely the constant polynomial 1), $\binom{m}{1} = m$ monomials of degree 1 (polynomials in one variable $x_1, x_2, \ldots, x_m$), $\binom{m}{2}$ monomials of degree 2 ($x_1x_2, x_1x_3, \ldots$ but not x_1^2). Therefore, there are

$$k = \sum_{i=0}^{r} \binom{m}{i} \tag{2.5}$$

such monomials of degree $\leq r$ and k is the dimension of the code: we can combine these monomials using binary coefficients to obtain a total of 2^k codewords; each corresponding to a polynomial of degree $\leq r$. The encoding evaluates the message polynomial at all 2^m assignments to the variables over $\mathbb{F}_2$ [HP03; Rob06], thus the length of the code is

$$n = 2^m \tag{2.6}$$

Formally, given a message polynomial P of degree $\leq r$, its encoding is $(P(a))_{a \in F_2^m}$. Let $V \subseteq \{1, \ldots, m\}$ be a set of variable indices, and let the monomial $R_S^V(X)$ of degree $|S|$ formed by the product of variables indexed by $S \subseteq V$ be:

$$R_{S \subseteq V} : \mathbb{F}_2^{|V|} \to \mathbb{F}_2 \tag{2.7}$$

$$R_{S \subseteq V}(X) = \prod_{i \in S} x_i \tag{2.8}$$

Let $[\![m]\!] = \{1, \ldots, m\}$ be the set containing the indices of all m variables. The evaluations of the monomials of degree $\leq r$ at all points:

$$\{(R_{S \subseteq [\![m]\!]}(a))_{a \in \mathbb{F}_2^m} \mid S \subseteq [\![m]\!], |S| \leq r\} \tag{2.9}$$

form a basis of the code. We can express any message polynomial P as a sum of monomials of degree less than r:

$$P(X) = \sum_{S \subseteq [\![m]\!], |S| \leq r} c_S \cdot R_{S \subseteq [\![m]\!]}(X) \tag{2.10}$$

where the coefficients $c_S \in \mathbb{F}_2$ determine whether a monomial, consisting of the variables whose indices are in S, is part of the message polynomial.

Majority Logic Decoding

RM codes admit a majority logic decoder [Irv53; VP06; ASY20]. We observe two properties. Let $S \subseteq V_m = \{1, \ldots, m\}$ be a set of indices of variables.

Lemma 2.2.2. For any set of variable indices S,

$$\sum_{a \in \mathbb{F}_2^{|S|}} R_{S \subseteq S}(a) = 1 \tag{2.11}$$

Proof. $R_{S\subseteq S}(X) = \prod_{i\in S} x_i$ is a monomial of degree $|S|$. Given that all variables occur in the monomial, the only assignment under which the polynomial evaluates to 1 is where all variables $\{x_i \mid i \in S\}$ are set to 1. For all other assignments the polynomial evaluates to 0, since at least one of the variables in the product is 0. Thus, the sum is 1. □

Lemma 2.2.3. For any proper subset $T \subset S$ of variable indices,

$$\sum_{a\in\mathbb{F}_2^{|S|}} R_{T\subseteq S}(a) = 0 \tag{2.12}$$

where the sum is computed in $\mathbb{F}_2$.

Proof. Recall that the only variable assignment for which $R_{T\subseteq T}(X) = 1$ are those where all variables $\{x_i \mid i \in T\}$ that occur in the monomial are set to 1: $a_T = \mathbf{1}^{|T|}$, where $a_T = Y$ means a is equal to Y on the coordinates in T. For the monomial with an expanded domain $R_{T\subseteq S}$, we find that we can pick any assignment $a \in \mathbb{F}_2^{|S|}$ for the variables that are not in T and obtain $R_{T\subseteq S}(a) = 1$, when $a_T = \mathbf{1}^{|T|}$. We have $\mu := 2^{|S\setminus T|}$ of these assignments. Since T is a proper subset of S, $\mu = 2^{|S|-|T|}$, $|T| < |S|$ and $\mu = 2^\ell$ for some $\ell \geq 1$. Since μ is a multiple of 2, the sum is 0 over $\mathbb{F}_2$. □

We now show a an identity that allows us to recover the coefficients c_S of P from a codeword.

Lemma 2.2.4. For any set of variable indices $S \subseteq \{1,\dots,m\}$ with $|S| = r$, a degree r polynomial P and assignments $b \in \mathbb{F}_2^{m-r}$

$$\sum_{a\in\mathbb{F}_2^m, a_{\bar{S}}=b} P(a) = c_S \tag{2.13}$$

where $\bar{S} = \{1,\dots,m\} \setminus S$ is the complement of S,

Proof. Let $P_{x_{\bar{S}}=b}$ be the polynomial obtained by substituting the $x_{\bar{S}}$ in P with the values in b for some fixed ordering of $\bar{S}$. Recall that P is a sum of monomials of degree $\leq r$:

$$P(X) = \sum_{V\subseteq[\![m]\!],|V|\leq r} c_V R_{V\subseteq[\![m]\!]}(X) \tag{2.14}$$

Then $P_{x_{\bar{S}}=b}$ is a sum of monomials with variables only in S:

$$P_{x_{\bar{S}}=b} : \mathbb{F}_2^r \to \mathbb{F}_2 \tag{2.15}$$

$$P_{x_{\bar{S}}=b}(X) = \sum_{V \subseteq [\![m]\!], |V| \leq r, V \subseteq S} c'_V \cdot R_{V \subseteq S}(X) \tag{2.16}$$

$$= \sum_{V \subseteq S} c'_V \cdot R_{V \subseteq [\![m]\!]}(X) \qquad \text{By } S \subseteq [\![m]\!] \text{ and } |S| = r \tag{2.17}$$

$$= c_S \cdot R_{S \subseteq S}(X) + \sum_{V \subset S} c'_V \cdot R_{V \subseteq S}(X) \qquad \text{Pull out term} \tag{2.18}$$

for some c'_V. Note that the coefficient $c'_S = c_S$ remains unchanged, since no variables are substituted in the corresponding monomial and no monomial has a degree greater than $|S| = r$. Therefore, no monomial has a superset of variable indices of S, whose variable substitution would result in a change to c'_S. Propagating the variables in this manner allows us to use Lem. 2.2.2 and 2.2.3 to obtain the result:

$$\sum_{a \in \mathbb{F}_2^m, a_{\bar{S}}=b} P(a) \tag{2.19}$$

$$= \sum_{a \in \mathbb{F}_2^r} P_{x_{\bar{S}}=b}(a) \tag{2.20}$$

$$= \sum_{a \in \mathbb{F}_2^r} \left(c_S \cdot R_{S \subseteq S}(a) + \sum_{V \subset S} c'_V \cdot R_{V \subseteq S}(a) \right) \qquad \text{By definition of } P_{x_{\bar{S}}=b} \tag{2.21}$$

$$= \left(\sum_{a \in \mathbb{F}_2^r} c_S \cdot R_{S \subseteq S}(a) \right) + \left(\sum_{a \in \mathbb{F}_2^r} \sum_{V \subset S} c'_V \cdot R_{V \subseteq S}(a) \right) \qquad \text{Distribute sum} \tag{2.22}$$

$$= c_S \cdot \left(\sum_{a \in \mathbb{F}_2^r} R_{S \subseteq S}(a) \right) + \left(\sum_{V \subset S} c'_V \cdot \sum_{a \in \mathbb{F}_2^r} R_{V \subseteq S}(a) \right) \qquad \text{Switch sums, Pull out constants} \tag{2.23}$$

$$= c_S \cdot 1 + \sum_{V \subset S} c'_V \cdot 0 \qquad \text{By 2.2.2, 2.2.3, } |S| = r, V \subset S \tag{2.24}$$

$$= c_S \tag{2.25}$$

□

Lemma 2.2.4 gives rise to an efficient decoding algorithm. Notice that the identity holds for any $b \in \mathbb{F}_2^{m-r}$. Additionally, we have 2^{m-r} such b. For each distinct b we obtain

a disjoint subset of $\mathbb{F}_2^m$: $\{a \in \mathbb{F}_2^m \mid x_{\bar{S}} = b\}$. The subsets are disjoint, since the b are distinct, thus they differ in at least one coordinate. We can perform a majority vote among these to decode the m^{th} order coefficients. To obtain the next coefficients we can reduce the decoding problem to that of a $R(1, m-1)$ code.

Hadamard Transform

Another way to decode RM codes is using the hadamard transform. The Hadamard transform decoder, or Green machine – due to its discovery by Green [MS77, pp.424-425] – allows for efficient decoding of first-order Reed-Muller codes [ASY20]. It uses the Fast Hadamard Transform (FHT).

Definition 2.2.7 (Hadamard transform)**.** Let the hadamard transform of dimension $2^m \times 2^m$ in Sylvester form be the linear map defined by [MS77, p.44]:

$$\mathbf{H}_m = \begin{cases} \mathbf{H}_1 \otimes \mathbf{H}_{m-1} = \begin{bmatrix} \mathbf{H}_{m-1} & \mathbf{H}_{m-1} \\ \mathbf{H}_{m-1} & -\mathbf{H}_{m-1} \end{bmatrix} & \text{if } m \geq 2 \\ \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} & \text{if } m = 1 \end{cases} \tag{2.26}$$

where $\otimes$ is the kronecker product.

This form of the hadamard transform is symmetric and satisfies the following equation closely relating it to its inverse:

$$\mathbf{H}_m \, \mathbf{H}_m^{\intercal} \, \mathbf{X} = \mathbf{H}_m \, \mathbf{H}_m \, \mathbf{X} = 2^m \, \mathbf{X} \tag{2.27}$$

The Hadamard transform is to Walsh functions what the discrete Fourier transform is to sinusoids. It decomposes a function into a superposition of Walsh functions, and can be computed efficiently in $\mathcal{O}(n \log n)$ time, instead of the naïve $\mathcal{O}(n^2)$, using the FHT, where $n = 2^m$. Rows and columns in a Hadamard matrix are correspond to codewords in $R(1, m)$. The FHT efficiently computes the correlations of a received word with each codeword $\mathbf{c}$ or its inverse $-\mathbf{c}$ [Hor07, p.42]. The decoder then decodes to the codeword has the highest correlation. Formally, let $\mathbf{f} \in \mathbb{F}_2^n$ be the received word. We convert the received word into a function with domain $\{-1, 1\}$: let $\mathbf{F} = (-1)^{\mathbf{f}} \in \{-1, 1\}^n$ be the vector obtained from $\mathbf{f}$ when replacing 0 by 1 and 1 by -1. Additionally, let $f(\mathbf{x}) = \mathbf{f}_{\mathbf{x}}$ denote the function yielding the $\mathbf{x}^{\text{th}}$ component of $\mathbf{f}$, where $\mathbf{x} \in \{-1, 1\}$. The hadamard transform $\hat{\mathbf{F}} = (\hat{F}(\mathbf{u}))_{\mathbf{u} \in \mathbb{F}_2^n}$ of $\mathbf{F}$ is [Hor07, p.56]:

$$\hat{F}(\mathbf{u}) = \sum_{\mathbf{v} \in \mathbb{F}_2^n} (-1)^{\langle \mathbf{u}, \mathbf{v} \rangle} F(\mathbf{v}) \tag{2.28}$$

We detail the Hadamard transform decoder in Section 4.3.3.

2.2.4 Cyclic Codes

Cyclic codes are linear codes with additional structure [HP03; Rob06]. The cyclic structure ensures, that cyclic shifts of a codeword are a codeword. More formally, let $\mathcal{C}$ be a linear $[n, k, d]$ code. If $(c_0, c_1, c_2, \ldots, c_{n-1})$ is a codeword in $\mathcal{C}$, then the cyclic shift by 1 $(c_1, c_2, \ldots, c_{n-1}, c_0)$ is also in $\mathcal{C}$. The coordinates are transformed by the map $i \mapsto i + 1 \pmod{n}$. Thus, by transitive closure, a cyclic code contains all cyclic shifts of a codeword. We can represent codewords in a cyclic code by polynomials in a polynomial ring modulo a polynomial of the form $x^n - 1$.

$$\mathcal{R}_n := \mathbb{F}_q[x]/\langle x^n - 1\rangle \tag{2.29}$$

Seen as such, a code $\mathcal{C}$ is isomorphic to the vector space $\mathbb{F}_q^n$, where the entries of a vector correspond to the coefficients of the polynomial. Vector addition adds the individual entries, and is equivalent to adding the two polynomials. The study of cyclic codes is closely linked to the study of ideals of $\mathcal{R}_n$.

Theorem 2.2.1. A cyclic code $\mathcal{C}$ is an ideal of $\mathcal{R}_n$.

Proof. A codeword $c \in \mathcal{C}$ satisfies the absorption property for any $p \in \mathcal{R}_n$:

$$p \cdot c = \sum_i p_i \cdot (c \cdot x^i) \tag{2.30}$$

The sum is in $\mathcal{C}$, since the cyclic shifts $c \cdot x^i$ are in the code, and the code is linear. $(\mathcal{C}, +)$ is a subgroup of $(\mathcal{R}_n, +)$, as $\mathcal{C}$ is a linear code. □

The study of ideals of $\mathbb{R}_n$ relies on factoring $x^n - 1$. The roots $x^n - 1$ are n^{th} roots of unity: For any n^{th} root of unity α, $\alpha^n = 1$, so the polynomial $x^n - 1$ is zero at α. Cyclic codes may be defined by their generator polynomial $g(x)$. Every codeword in $\mathcal{C}$ is a multiple of g in $\mathcal{R}_n$. Cyclic codes are generated by a circulant generator matrix. A circulant matrix is a matrix where each row is obtained from a previous one by a cyclic shift by one position. Circulant matrices can be used to represent the multiplication of polynomials. Thus, the generator matrix contains the cyclic shifts of the generator polynomial in its rows.

2.2.5 Bose-Chaudhuri-Hocquenghem Codes

BCH codes are a family of cyclic codes [HP03; Rob06]. The BCH-bound gives a lower bound on the minimum distance of a cyclic code.

Theorem 2.2.2. Let $\mathcal{C}$ be a cyclic code of length n and minimum distance d over $\mathbb{F}_q$. If the generator polynomial has $\delta - 1$ roots at consecutive powers of a primitive element $\alpha \in \mathbb{F}_q$, then $d \geq \delta$ (stated without proof [HP03, p.151]).

The BCH bound leads to the definition of the BCH codes, exploiting the stated property to maximize the minimum distance. The dimension of a cyclic code is $n - \deg g$. Thus, we would like to minimize the degree of g. The BCH bound tells us that at the same time we want g to have as many roots at consecutive powers of α as possible. For a given number of roots $\delta - 1$, we define the generator polynomial g:

$$g(x) = \operatorname{lcm}(M_{\alpha^i}(x), M_{\alpha^{i+1}}(x), M_{\alpha^{i+2}}(x), \ldots, M_{\alpha^{i+\delta-2}}(x)) \tag{2.31}$$

for some i, where $M_r(x)$ is the minimal polynomial[1] with a root at r: $M_r(r) = 0$. Choosing the least common multiple ensures that the degree of g is minimized, while g has $\delta - 1$ consecutive roots.

2.2.6 Reed-Solomon Codes

RS codes are a subfamily of the BCH codes [HP03; Rob06]. In RS codes the length of the code n is tied to the size of the alphabet q: $n = q - 1$. The α in the factors $x - \alpha$ of the polynomial $x^n - 1$ are the n^{th} roots of unity. When $n = q - 1$ the α are all contained in $\mathbb{F}_q$: Given a primitive n^{th} root of unity α, it generates the multiplicative subgroup: $|\{\alpha^i \mid i \in [\![q-1]\!]\}| = q - 1$ and $\alpha^{q-1} = 1$. Thus, the elements α^i generated by the generator α are n^{th} roots of unity: $(\alpha^i)^n = (\alpha^n)^i = (1)^i = 1$. Since all n^{th} roots of unity are in fact contained in $\mathbb{F}_q$ we can write g simply as

$$g(x) = (x - \alpha^b)(x - \alpha^{b+1}) \ldots (x - \alpha^{b+\delta-2}) \tag{2.32}$$

for some b and the desired minimum distance δ.

2.2.7 Quasi-Cyclic and Double Circulant Codes

Quasi-Cyclic (QC) codes are a generalization of cyclic codes. A code is QC if there exists an integer s such that every cyclic shift by s of any codeword yields another codeword. Such a code is a QC code of *index* s. Cyclic codes are QC codes for $s = 1$ [HP03, p.132]. We concentrate on double circulant codes, as they are used in HQC. Double circulant codes are equivalent to a QC code of index $s = 2$ [MS77, p.506]. We further restrict our study of double circulant codes for which the generator matrix $\mathbf{G}$ has the following form:

$$\mathbf{G} = \left[\mathbf{I}_k \mid \mathbf{A}\right] \tag{2.33}$$

where $\mathbf{A}$ is a circulant $k \times k$ matrix. $\mathbf{G}$ generates a $[2k, k]$ code. $\mathbf{G}$ can be transformed into a QC code by reordering the columns in the following order: $1, k+1, 2, k+2, \ldots$.

[1] polynomial of least degree

2.2.8 Quasi-Cyclic Syndrome Decoding

HQC security reduces to the Quasi-Cyclic Syndrome Decoding (QCSD) problem. They introduce the QCSD distribution yielding the syndrome of a randomly sampled vector with respect to a random QC code. We omit some details and refer to [Car+20] for a more in-depth treatment.

Definition 2.2.8 (s-QCSD Distribution)**.** The s-QCSD(n,w) distribution chooses uniformly at random a parity-check matrix H of a QC code of index s and a random vector $\mathbf{x}$ of weight w and outputs $(H, H\mathbf{x}^\intercal)$ – H and the syndrome of $\mathbf{x}$.

We can construct a computational and decisional QCSD problem from this distribution. The computational problem asks to recover $\mathbf{x}$ from $(H, \mathbf{y})$ sampled from the QCSD distribution. The decisional variant asks to distinguish samples from the QCSD distribution from a uniform distribution.

2.3 Code-Based Cryptography

Problems in coding theory may be used to construct cryptographic primitives such as One-Way Functions (OWFs) [Lö15]. However, code-based cryptography ventures beyond the limits of minicrypt [Imp95] and allows for the construction of Trapdoor One-Way Functions (T-OWFs) [Lö15]. These may then be used to construct a multitude of cryptographic primitives in cryptomania such as PKE schemes or KEMs.

2.3.1 The McEliece Cryptosystem

The McEliece T-OWFs is the first publicly known use of coding theory for asymmetric cryptography [Rob78]. McEliece bases its security on the hardness of decoding a general linear code. The problem of decoding a general linear code was shown to be NP-complete [BMT78], giving some level of confidence, that the cryptosystem may be secure. However, as the Merkle-Hellman knapsack cryptosystem demonstrates, basing a cryptosystem on NP-complete problems does not guarantee security [Sha82].

Definition 2.3.1. McEliece Cryptosystem [Lö15; Rob78] The McEliece cryptosystem is an algorithm triple $\Delta = (\text{KeyGen}, \text{Enc}, \text{Dec})$ where:

- Let $\mathcal{C}$ be a randomly chosen linear code from a family of efficiently decodable codes, and $\mathbf{G} \in \mathbb{F}_q^{k \times n}$ be a generator matrix of $\mathcal{C}$. Additionally, let $\mathbf{S}$ be an random invertible matrix and $\mathbf{P}$ a permutation matrix.

  $\text{KeyGen}(1^\lambda) = (\mathsf{sk}, \mathsf{pk}) = ((\mathbf{S}, \mathbf{G}, \mathbf{P}), (\mathbf{SGP}, w))$

  where λ is the security parameter, and $\mathbf{G}$ can correct at least w errors.

- $\text{Enc}(1^\lambda, \mathsf{pk} = (\mathbf{SGP}, w), \mathbf{m}) = \mathbf{mSGP} + \mathbf{e}$

  where $\mathbf{e} \leftarrow\!\!\$\, \mathbb{F}_q^n$ is a random error of weight w.

- $\mathrm{Dec}(1^\lambda, \mathsf{sk}, \mathbf{c} = \mathbf{mSGP} + \mathbf{e}) = \mathrm{Decode}(\mathbf{cP}^{-1})\mathbf{S}^{-\mathbf{1}}$

The goal of the $\mathbf{S}$ and $\mathbf{P}$ matrices is to transform the efficiently solvable decoding problem into a random looking instance of the problem, which cannot be solved efficiently in general.

$$\mathbf{c} = \underbrace{(\mathbf{m}\hat{\mathbf{G}} + \mathbf{e})}_{\text{hard}} = (\mathbf{m}(\mathbf{SGP}) + \mathbf{e}) = ((\mathbf{mS})\mathbf{GP} + \mathbf{e}) \tag{2.34}$$

However, the intended recipient can transform the ciphertext using the trapdoor information $\mathbf{S}$ and $\mathbf{P}$ into an efficiently solvable instance:

$$\mathbf{cP}^{-1} = (\mathbf{m}\hat{\mathbf{G}} + \mathbf{e})\mathbf{P}^{-1} = \mathbf{mSGPP}^{-\mathbf{1}} + \mathbf{eP}^{-\mathbf{1}} = (\mathbf{mS})\mathbf{G} + \mathbf{eP}^{-\mathbf{1}} = \underbrace{\hat{\mathbf{m}}\mathbf{G} + \hat{\mathbf{e}}}_{\text{easy}} \tag{2.35}$$

Note that the McEliece cryptosystem does not fulfill strong security notions such as IND-CCA [KL14] that we now require of modern PKEs schemes. It is easy to show that the scheme is not IND-CPA [KL14] secure: The adversary may send $\mathbf{m}_0, \mathbf{m}_1 \leftarrow\!\$\, \mathbb{F}_q^k$ to the challenger, and receives $\mathbf{c}_b = \mathbf{m}_b\mathbf{SGP} + \mathbf{e}$. The adversary then computes the weight of $\mathbf{c}_b + \mathbf{m}_0\mathbf{SGP}$ and $\mathbf{c}_b + \mathbf{m}_1\mathbf{SGP}$. In all but negligibly many cases one of these weights will be exactly w, since the error $\mathbf{e}$ added to the ciphertext is of weight w. The one with weight w corresponds to the message that was encrypted. Fortunately, the scheme may be considered OW-CPA secure for suitable parameter choices [KI03]. Transformations that forge an IND-CCA secure KEM from a OW-CPA secure scheme exist [HHK17].

The McEliece T-OWF did not gain much popularity initially, as more efficient alternatives existed at the time. However, with the rise of post-quantum cryptography, code-based schemes garnered newfound interest, as there are no known efficient attacks in a quantum-adversary setting against McEliece. Code-based schemes are now among the leading contenders in the NIST effort for standardization of post-quantum cryptography.

2.3.2 The Niederreiter Cryptosystem

The Niederreiter cryptosystem is similar to the McEliece cryptosystem but uses the dual code generated by the parity check matrix $\mathbf{H} \in \mathbb{F}_q^{(n-k)\times n}$ of $\mathcal{C}$ [Har86]. The sender maps the message to an error vector $\mathbf{e} \in \mathbb{F}_q^n$ of a fixed weight w. They then compute the ciphertext as the syndrome $\hat{\mathbf{H}}\mathbf{e}^\intercal$, where the public parity check matrix $\hat{\mathbf{H}} = \mathbf{SHP}$. The receiver decrypts the message using a syndrome decoding algorithm. The Classic McEliece KEM [Alb+20] is based on the Niederreiter construction. While not very efficient, as it has a large public-key, its security is less debatable when compared to modifications using different codes, since the methods used in the Niederreiter cryptosystem have resisted attacks for several decades. Unfortunately, many modifications meant to increase the efficiency of these schemes by replacing the codes used, turned out to be insecure [Lö15].

2.4 Side-Channel Attacks

Whenever the concrete execution model deviates from an abstract model of computation in a way that is observable by an adversary, the deviation may constitute a side-channel. We often prove protocols and schemes secure under an abstract model of computation, where every agent is a black box and does not emit any information besides the result of a computation. However, in the real world adversaries easily have access to the time a computation takes, with different levels of precision. We provide an incomplete list of side-channels that are commonly exploited in attacks on cryptographic implementations:

1. Electromagnetic radiation [Gen+16]
2. Power consumption [KJJ99]
3. Timing [Koc96; AFP13; AP16; BB05]
4. Cache timing [BH09; ASK06; GB+16; Ber05]

An additional category of attacks that exploits the physical nature of systems are fault attacks. These attacks actively tamper with the physical implementation by, for example, manipulating the power supplied to the target. Hardening implementations against these attacks presents a tremendous challenge and protections against these attacks are often only achieved with respect to an adversary with restricted capabilities [Bra+20]. Often these attacks assume powerful adversaries with physical access to the victim's hardware [Kim+14; BE+06]. However, researchers have demonstrated the feasibility of such attacks remotely via the network [Lip+20] or through software [Mur+20], so one cannot rule these threats out when physical access is assumed secure.

2.4.1 Timing Attacks

In 1996 Kocher [Koc96] published the first publicly known complete break of modern cryptosystems using timing side-channel information. Kocher discovered that the computation time of various implementations of cryptosystems is dependent upon secret information such as key material or plaintexts. Kocher stated that intuition would suggest that one cannot obtain a significant amount of information about secrets using timing side-channels. However, as Kocher's results show timing side-channels can lead to full recovery of secrets. We illustrate how timing side-channels can leak secrets in vulnerable implementations using a non-constant-time comparison function of two vectors $\mathbf{a}$ and $\mathbf{b}$ shown in Algorithm 2.1. The algorithm keeps track of whether all elements of the two vectors have been equal up to the current index i. As soon as it encounters a coordinate where the vectors differ, it exits the loop early. Notice, that the runtime of this algorithm depends on the number of elements in the longest common prefix of $\mathbf{a}$ and $\mathbf{b}$. Assume now that we are given access to the oracle in Eq. (2.36) taking an adversarially chosen input $\mathbf{a}$.

Algorithm 2.1: memcmp

Input: Vectors $\mathbf{a}$ and $\mathbf{b}$ of length n with elements from $\{0,1\}^8$.
Result: $\mathbf{a} \stackrel{?}{=} \mathbf{b}$

```
1 i = 1
2 while i ≤ n do
3 |   if a_i ≠ b_i then
4 |   |   return false
5 |   end
6 |   i = i + 1
7 end
8 return true
```

$$\mathcal{O}_{\texttt{memcmp}}(\mathbf{a}) = \texttt{memcmp}(\mathbf{a}, \mathbf{secret}). \tag{2.36}$$

Then we can recover the secret by first enumerating every possibility for the first coordinate. For each possibility we measure the time it takes for the oracle to respond to our query. For one of the options the oracle will return an answer after a longer duration, meaning that we guessed the first coordinate correctly. We can then continue the attack with the second coordinate, keeping the first coordinate fixed to the obtained value. Thus we can recover the secret much quicker. Notice, that this constitutes an exponential speedup: given only access to the result of the computation and applying brute force, we would need $\mathcal{O}\left((2^8)^n\right) = \mathcal{O}(2^{8n})$ guesses to obtain the secret. However, using timing information we can guess each byte individually and only have to perform $\mathcal{O}(2^8 n) = \mathcal{O}(n)$ guesses. In practice, due to noise, we would have to repeat each query several times to obtain a good estimate of the actual runtime. However, this only constitutes a constant factor slow-down. While this timing side-channel may seem obvious, the use of precisely this algorithm has been exploited in a recent attack on post-quantum cryptographic schemes [GJN20]. Various sources of timing side-channels have been discovered. These include, but are not limited to [Tima]:

Secret dependent accesses

For example, caching effects can cause a varying execution time: accessing an array at a secret index will take a differing amount of time, depending on whether the accessed memory region is cached or not. This can reveal information about the secret index. A program running with normal user privileges can typically not ensure, that some data is retained in cache for the entire duration of a computation. This is due to kernels preempting the program, and modifying the cache state by running other programs. Even if one could ensure that an entire array is kept in cache for the entirety of the execution, other hardware effects would still cause variable timings: on some CPUs a load from L1 cache can take more time if a store was recently issued to a cache-line associated with the same memory address [Ber05].

Conditional jumps based on secret data
Jumping to different locations depending on secret data can reveal information about the secrets since the execution time of the two branches may differ. If that is not the case, one of these locations may be in the instruction cache and not the other.

Variable-time CPU instructions operating on secret data
Some CPU instructions, such as division instructions, take a varying number of cycles depending on the input [Agn].

While divisions are seldomly used in cryptographic implementations, caching is a renowned problem for software implementations of AES, for example [Ber05]. Conditional branches often also present issues, for example when the algorithm loops a different number of times depending on a secret. Note that when we talk about secrets we not only mean the secrets itself (such as keys or messages), but also values derived from these secrets. This is of particular importance for our investigations. In general, any value for which there is an information flow from a secret to that value is considered a secret.

2.4.2 Identifying Timing Leaks

Identifying timing leaks can be performed automatically to some extent [Imp; Tima]. However, depending on the precision of the analysis, these automated analyses can also turn up false positives, decreasing their usefulness. When we encounter a positive result we have to further investigate, whether the test outcome is justified. Additionally, these tests cannot exclude the existence of a timing leak with certainty. Any single empirical test that one can perform essentially assumes a specific adversary strategy [Sta19]. Approaching the problem empirically one can, for example, look for a correlation between CPU cycles and selected properties of secrets, such as the weight of the error term used in HQC. The Test Vector Leakage Assessment (TVLA) [Bec+] can be used to test whether a program emits the same side-channel behavior for every input. The method is mainly employed for physical side-channels. The dudect [Rep20] tool is akin to TVLA tests in that it uses statistical methods and empirical data to detect timing variations in crypto code. A major aspect of such leakage detection tools is to decide which test vectors to use. These must be tailored to the specific algorithm at hand.

Formal analysis tools such as ct-verif [Alm+16] or FlowTracker [RQPA16] can verify whether a given program has flows from secrets to branches or lookups. Both of these tools base their analyses on intermediate representations close to the ones produced by clang when compiling C code. Unfortunately, ct-verif left SSE instructions, which HQC makes heavy use of, up for future work. Additionally, we could not find any updates to the implementation in their GitHub repository that would add support for SSE instructions [Imd]. Therefore the tool may refuse analysis or is forced to assume incorrect or over-approximated behavior and thus introduces unsoundness or incompleteness. Such holes in the analysis are common in formal analysis tools, and are hard to get rid of.

Removing them often requires modeling the behavior of the underlying ISA with great precision.

Ctgrind [Lan10] is a valgrind patch that uses memcheck's ability to detect uses of uninitialized values to detect secret dependent branches or lookups. The authors argue that cryptographic code that is not trivially broken is mostly straight line code, and thus a single run would obtain a large coverage. According to the authors of dudect, similar results can also be obtained using a stock valgrind version, by leaving secrets uninitialized. This methodology was used by Bernstein to dispute HQC's constant time claims [Hqcd]. Bernstein's comment also notes that there are many other paths noted by the tool as potential leaks.

TIMECOP [Tima] uses valgrind to test a range of cryptographic implementations for timing side-channel leakage. However, HQC is not contained in suite. Further, all except five of the many schemes listed exhibit timing variations according to the analysis. This may mean that there are many false positives.

In this thesis we use a custom method for identifying timing leaks, which we describe in detail in Section 4.1.

2.4.3 Countermeasures

A countermeasure makes exploitation of a side-channel ineffective or practically impossible. To achieve this there are several possible approaches:

1. Reducing channel capacity
2. Removing the dependency of the side-channel information on secrets

Implementing algorithms in a constant-time manner is one way to remove the dependency of the side-channel information on secrets.

Definition 2.4.1 (Constant-time)**.** An algorithm is *constant-time*, when its runtime is independent of secrets.

Note, that this definition does not strictly say, that the computation time must be some constant. It merely implies, that the distribution of timings must be independent of the secrets. The algorithm may leak any non-secret information through timings. One way to provably ensure an algorithm is constant-time is to restrict oneself to a certain subset of instructions that are known to execute in constant-time when handling secrets [Zin+17]. However, this assumes perfect knowledge of the behavior of the underlying ISA and will reject safe programs when variable-time instructions are used to handle non-secret data, or when the set of allowed instructions is chosen too conservatively. Fortunately, there are generic countermeasures for most violations of constant-time principles.

1. To avoid a conditional branch dependent upon secrets, we execute both paths unconditionally and then mask the desired result in using constant time operations.

2. To avoid lookups into an array using secret dependent indices, we can scan the entire array, and mask in the desired value when we are at the correct position.

   We show a concrete implementation of this method in Section 7.2.

3. To avoid variable-time instructions, we must re-implement their functionality in terms of other constant-time instructions.

   In Section 7.1.1 we discuss how, under certain circumstances, we can replace a divide instruction by other instructions, making an algorithm constant-time.

Lamentably, these generic countermeasures can also incur a heavy performance hit. It is also possible that the transformation cannot be applied, if the algorithm's worst-case runtime is unbounded. In Chapter 7 of this thesis we will discuss countermeasures addressing this issue. We demonstrate these using the example of the algorithm identified as the cause of the timing variation which we discover in HQC.

CHAPTER 3

The HQC Scheme and Prior Timing Attacks

In this chapter we introduce the HQC scheme and the state of the art in attacking HQC using timing side-channels embedded in its C implementation.

3.1 Overview of the HQC Scheme

Hamming Quasi-Cyclic (HQC) [Agu+16; Car+20] is a code-based post-quantum CCA secure KEM. The HQC framework from which HQC is introduced in *Efficient Encryption from Random Quasi-Cyclic Codes* [Agu+16]. The scheme uses two kinds of codes: a publicly known ECC $\mathcal{C}$ generated by $\mathbf{G} \in \mathbb{F}_2^{k \times n}$ and a random double-circulant code. HQC's security is reduced to problems related to the hardness of decoding random quasi-cyclic codes in the Hamming metric.

A summary of the algorithms of the HQC PKE scheme can be found in Fig. 3.1. The HQC PKE consists of four functions. The setup function selects the most efficient parameters that satisfy the desired security level given by the security parameter λ.

n the size of elements in the polynomial factor ring $\mathcal{R}_n$

k the size of messages encoded by the publicly known ECC code

δ the minimum number of errors the publicly known code $\mathcal{C}$ can correct

ω weight of vectors in the private key

ω_e, ω_r
weight of vectors used during encryption

The key generation function generates a public key (pk) and secret key (sk). The public key may be used to encrypt a message using the encryption function and obtain a ciphertext. Given the respective secret key, the recipient can then decrypt the ciphertext. Unlike other code-based schemes such as McEliece or Niederreiter, HQC does not try to hide the structure of a secret code in its public key. Instead, the key generation part of HQC generates a syndrome $\mathbf{s}$ of a random low weight vector $(\mathbf{x}, \mathbf{y})$ with respect to a random QC code with parity check matrix $\left[\mathbf{I} \mid \mathbf{h}\right]$. Encryption generates another syndrome $\mathbf{u}$ of a random low weight vector $(\mathbf{r_1}, \mathbf{r_2})$. It then encodes the message $\mathbf{m}$ using the generator matrix $\mathbf{G}$ of a publicly known code. This encoding is then hidden using the product $\mathbf{s} \cdot \mathbf{r_2}$ and an additional error $\mathbf{e}$ is added. This hidden encoding of $\mathbf{m}$ forms $\mathbf{v} := \mathbf{mG} + \mathbf{s} \cdot \mathbf{r_2} + \mathbf{e}$. To decrypt, we compute:

$$\begin{aligned} & \mathbf{v} - \mathbf{u} \cdot \mathbf{y} && (3.1)\\ = {} & (\mathbf{mG} + \mathbf{s} \cdot \mathbf{r_2} + \mathbf{e}) - (\mathbf{r_1} + \mathbf{h} \cdot \mathbf{r_2}) \cdot \mathbf{y} && (3.2)\\ = {} & \mathbf{mG} + (\mathbf{x} + \mathbf{h} \cdot \mathbf{y}) \cdot \mathbf{r_2} - (\mathbf{r_1} + \mathbf{h} \cdot \mathbf{r_2}) \cdot \mathbf{y} + \mathbf{e} && (3.3)\\ = {} & \mathbf{mG} + \mathbf{x} \cdot \mathbf{r_2} - \mathbf{r_1} \cdot \mathbf{y} + \mathbf{e} && (3.4) \end{aligned}$$

The result is sent to the decoder. The decoder has to correct the error $\mathbf{x} \cdot \mathbf{r_2} - \mathbf{r_1} \cdot \mathbf{y} + \mathbf{e}$. The weight of this error is low enough with high probability. The public code and associated decoder should be chosen such that they can correct this error with probability very close to 1. For HQC the DFR is less than 2^{-64}.

The IND-CPA security of the HQC PKE is reduced to a QCSD problem. In particular, through a series of game-hops the encryption of a message $\mathbf{m}_1$ is transformed to an encryption of another message $\mathbf{m}_2$ [Sho04]. In each transition the authors of HQC prove that an adversary who could distinguish whether they are playing either game could also break the QCSD problem. Thus, under the assumed hardness of the QCSD problem, the HQC PKE is IND-CPA secure [Agu+16; Car+20].

The HQC KEM is constructed from a PKE using the FO or Hofheinz Hövelmanns Kiltz (HHK) transformation [HHK17]. The HHK transformation can be applied to an OW-CPA secure PKE scheme to obtain an IND-CCA secure KEM. The HQC KEM again consists of four functions. These functions are listed in Fig. 3.2. The setup and key generation function are identical to the functions in the PKE. The "Encaps" or encapsulation function takes a public key and returns a tuple consisting of the shared key K and a ciphertext tuple (c, d). The "Decaps" or decapsulation function takes the respective secret key and a ciphertext to decrypt and returns the same shared key K or aborts if a decryption failure occurred. To construct the KEM from the PKE let $\mathcal{G}, \mathcal{K}, \mathcal{H}$ be independent hash functions. To encapsulate we sample a random message $\mathbf{m}$. We encrypt this message using randomness θ derived from the message using the hash function $\mathcal{G}$. This yields the ciphertext c. We derive the shared key K from $\mathbf{m}$ and c as $\mathcal{K}(\mathbf{m}, c)$. Finally, we compute a hash $d = \mathcal{H}(\mathbf{m})$ of the message. We output the shared key K and the ciphertext (c, d).

Setup(1^λ)

1 Select global parameters

2 $param = (n, k, \delta, \omega, \omega_r, \omega_e)$.

3 **return** *param*

KeyGen(param)

1 $\mathbf{h}, \mathbf{x}, \mathbf{y} \leftarrow\$ \mathcal{R}$ with $\omega(\mathbf{x}) = \omega(\mathbf{y}) = \omega$

2 $\mathsf{sk} = (\mathbf{x}, \mathbf{y})$

3 $\mathsf{pk} = (\mathbf{h}, \mathbf{s} = \mathbf{x} + \mathbf{h} \cdot \mathbf{y})$

4 **return** $(\mathsf{pk}, \mathsf{sk})$

Encrypt(pk, $\mathbf{m}$)

1 $\mathbf{e}, \mathbf{r_1}, \mathbf{r_2} \leftarrow\$ \mathcal{R}$ with $\omega(\mathbf{e}) = \omega_e$ and $\omega(\mathbf{r_1}) = \omega(\mathbf{r_2}) = \omega_r$

2 $\mathbf{u} = \mathbf{r_1} + \mathbf{h} \cdot \mathbf{r_2}$

3 $\mathbf{v} = \mathbf{mG} + \mathbf{s} \cdot \mathbf{r_2} + \mathbf{e}$

4 **return** $c = (\mathbf{u}, \mathbf{v})$

Decrypt($\mathsf{sk} = (\mathbf{x}, \mathbf{y})$, $c = (\mathbf{u}, \mathbf{v})$)

1 **return** $\mathcal{C}.\text{Decode}(\mathbf{v} - \mathbf{u} \cdot \mathbf{y})$

Figure 3.1: HQC PKE scheme consisting of a parameter generation, key generation, encryption and decryption function.

In the decapsulation we decrypt the ciphertext c to obtain a message $\mathbf{m}'$. Note that the PKE scheme is only IND-CPA secure, so an adversary could malleate the message such that $\mathbf{m} \neq \mathbf{m}'$. To verify that the ciphertext was generated honestly, we first perform a re-encryption of the message $\mathbf{m}'$ using randomness θ' derived from $\mathbf{m}'$. Then, we check whether the re-encrypted ciphertext matches the received ciphertext and whether the sent hash d matches the hash of the decrypted message $\mathbf{m}'$. These steps transform the scheme from an IND-CPA secure PKE to a QKEM$_m^\perp$ [HHK17].

HQC uses the public code generated by $\mathbf{G}$ to correct the errors introduced during encryption and decryption. For this purpose two codes have been introduced: either a BCH code combined with a repetition code or a RS code combined with a RM code. The BCH variant first encodes a message with a BCH code and then repeats each symbol multiple times. The RMRS variant encodes a message with a RS code and then encodes

Setup(1^λ)

1 **return** PKE.Setup(1^λ)

KeyGen(param)

1 **return** *PKE*.KeyGen*(param)*

Encaps(pk)

1 $\mathbf{m} \leftarrow\$ \mathbb{F}_2^k$

2 $\theta = \mathcal{G}(\mathbf{m})$

3 $c = \text{PKE.Encrypt}(\mathsf{pk}, \mathbf{m}; \theta)$

4 $K = \mathcal{K}(\mathbf{m}, c)$

5 $d = \mathcal{H}(\mathbf{m})$

6 **return** $(K, (c, d))$

Decaps($\mathsf{sk} = (\mathbf{x}, \mathbf{y})$, $(c = (\mathbf{u}, \mathbf{v}), d)$)

1 $\mathbf{m}' = \text{PKE.Decrypt}(\mathsf{sk}, c)$

2 $\theta' = \mathcal{G}(\mathbf{m}')$

3 $c' = \text{PKE.Encrypt}(\mathsf{pk}, \mathbf{m}'; \theta')$

4 **if** $c \neq c' \vee d \neq \mathcal{H}(\mathbf{m}')$ **then**

5 | abort

6 **end**

7 **return** $K = \mathcal{K}(\mathbf{m}', c)$

Figure 3.2: HQC KEM consisting of a parameter generation, key generation, encapsulation and decapsulation function.

each byte with a duplicated RM code. We detail this procedure further in Section 4.3.3.

3.2 Prior Timing Attacks on HQC

To date we could identify three papers that present timing-attacks on the HQC scheme [WT+19; PT19; GJN20]. All of them are able to recover the secret key from timing leakage. Two of them [WT+19; PT19] concern themselves with a non-constant time BCH decoder. The remaining one stems from using a non-constant time comparison [GJN20].

The side-channel discovered by Guo, Johansson, and Nilsson [GJN20] resembles the side-channel we discover in this thesis the most. The decapsulation algorithm of an HHK transformed scheme not only decrypts a given ciphertext, but also re-encrypts the decrypted message using derived randomness. Then, the resulting ciphertext is compared to the received ciphertext. If these mismatch, the ciphertext is rejected. The discovery by Guo et al. lies in the ciphertext comparison: if this step is not implemented in constant time, we can forge an algorithm. This algorithm malleates a ciphertext and times the decapsulation to deduce information on the secret key. The attack could be used on FrodoKEM [Nae+20], LAC [Lu+19], HQC and potentially other schemes. The authors estimate the attack to require 2^{30} decapsulations for FrodoKEM. They state that their attack can be adapted to HQC, however do not detail their procedure, claiming that it is similar to their attack on LAC. The underlying issue they exploit has been resolved in the PQClean [Pqc] implementation of HQC.

Recently, HQC's supposedly constant time implementation was found to contain a Galois field multiplication using lookup tables [Hqcd], which could potentially leak timing-information. According to the authors of the scheme, they failed to replace the non-constant time version when updating their implementation. The authors have now delivered a new constant time version as part of an update to their implementation. Prior to this incident, HQC made use of a non-constant time decoder. Use of the non-constant time decoder is exploitable. Wafo-Tapa, Bettaieb, Bidoux, Gaborit, and Marcatel [WT+19] and Paiva and Terada [PT19] present timing attacks against HQC. Both exploit that the BCH decoder is not implemented in a constant time manner. The BCH decoder takes more time when it has to correct more errors [PT19]. This enables a chosen ciphertext attack recovering part of the secret key, which can then be used to compute the remaining part.

HQC's CPA secure version is attacked in [WT+19], the CCA secure version in [PT19]. As a countermeasure, Wafo-Tapa, Bettaieb, Bidoux, Gaborit, and Marcatel additionally present a constant time BCH decoder.

Many code-based and lattice-based cryptosystems have a non-zero DFR. For KEMs this means that the decapsulation fails with some non-zero probability, even for ciphertexts generated honestly using the encapsulation algorithm. If this DFR is non-negligible, one may be able to deduce information about the secret key or plaintext, by recording which ciphertexts can be decapsulated by a decapsulation oracle. Therefore, designers strive to

Table 3.1: HQC parameter set aliases using our naming scheme.

Submission	Release	Instance	Code Type	Alias
2020-05-29	6 [Hqca]	hqc-128	BCH	hqc-r6-bch-128
		hqc-192		hqc-r6-bch-192
		hqc-256		hqc-r6-bch-256
		hqc-rmrs-128	RMRS	hqc-r6-rmrs-128
		hqc-rmrs-192		hqc-r6-rmrs-192
		hqc-rmrs-256		hqc-r6-rmrs-256
2020-10-01	7 [Hqcc]	hqc-128	RMRS	hqc-r7-rmrs-128
		hqc-192		hqc-r7-rmrs-192
		hqc-256		hqc-r7-rmrs-256
2021-06-06	7 [Hqcb]	hqc-128	RMRS	hqc-r8-rmrs-128
		hqc-192		hqc-r8-rmrs-192
		hqc-256		hqc-r8-rmrs-256

obtain a negligible DFR. However, a low DFR alone is not sufficient: error amplification can re-enable these kinds of attacks [NJW18].

Paiva and Terada [PT19] base their attack on prior work in [GJS16]. In this paper the authors publish their discovery of a relationship between decoding errors and the distance spectrum of the secret key in QC-Moderate-Density Parity-Check (MDPC) schemes. They define the distance spectrum as the cyclic distances between any two ones, and are able to recover this spectrum using a reaction oracle, revealing whether a given ciphertext decrypts correctly, or not. They show an algorithm enabling recovery of the secret key using its spectrum. The side channel identified by Paiva and Terada enables recovery of the spectrum.

The attack by Wafo-Tapa, Bettaieb, Bidoux, Gaborit, and Marcatel [WT+19] distinguishes whether the BCH decoder in HQC has to correct either exactly 0 or 1 errors. Using this primitive they devise an attack that can identify the location of the ones in the secret key. The attack uses the additional repetition code that is decoded before the BCH code.

3.3 Parameter Sets

There have been 7 releases of the HQC submission package to date. We concentrate on 2 releases, in particular the 6^{th} release from 2020-05-29 and the 7^{th} release from 2020-10-01. The 6^{th} release [Hqca] of HQC included parameter sets for both the BCH and RMRS version. These versions were called hqc-128 and hqc-rmrs-128 respectively for 128 bit security. In the 7^{th} release [Hqcc], the BCH version was deprecated and the hqc-rmrs version is now called just hqc. Since we deal with both the BCH and RMRS versions in this thesis, we break with the naming convention suggested by the authors of HQC. A similar naming convention break was also done by PQClean [Pqc] who retain the

hqc-rmrs name even for the 7$^{\text{th}}$ release. Additionally we append the release number to HQC. For brevity we may omit the "hqc-" prefix. Our naming convention is consistent with respect to the code type and can uniquely identify a parameter set in the examined HQC submissions.

All parameter sets and their aliases that are relevant in this thesis are listed in Table 3.1.

CHAPTER 4

A Novel Timing Side-Channel Attack

In this chapter we show how all to-date published implementations of HQC are vulnerable to another timing side-channel attack. We will demonstrate this vulnerability using the optimized implementations of hqc-r6-bch-128 and hqc-r8-rmrs-128 for x86 based systems. First, in Section 4.1, we identify a timing-variation in the HQC KEM. Then, in Section 4.2 and 4.3 we present two attacks exploiting this timing-variation to recover the key. Additionally, in Section 4.4, we identify a flaw in the analysis of previous side-channel attacks against HQC. This flaw would prevent an attacker from successfully recovering the secret key for some keys. We propose a solution to address this issue. With these lessons learned we evaluate the attacks empirically in Chapter 5. Then, we analyze the side-channel formally in Chapter 6 to demonstrate that the timing-variation is exploited optimally, with respect to the choice of the message. To mitigate the issues present in the HQC implementation we propose and analyze several countermeasures in Chapter 7. Finally, in Chapter 8, we show how a structurally similar timing-variation is present in BIKE [Ara+20].

4.1 Discovery of the Timing-Variation

We now detail the way in which we discovered that yet another previously undiscovered timing side-channel hides in the optimized HQC implementation.

4.1.1 Measuring Cycles

For empirical tests we need to measure the execution time of the decapsulation function. To this end we measure the number of cycles a piece of code takes to execute. Another interesting metric to measure is the number of instructions executed. It can yield very precise measurements and eradicate the need for measuring a piece of code multiple times. However it will also exclude some causes of side-channels: division instructions

```
cpuid
rdtsc
; code to measure
rdtscp
cpuid
```

Listing 1: x86 assembly for measuring execution time of code in cycles.

take a varying number of cycles depending on their input, but would only show up as a single executed instruction. Additionally, we are not aware of any ways to measure this side-channel remotely. Therefore we use cycles as our surrogate for time. Following Intel recommendations [Gab10] we use the structure shown in Listing 1. This structure yields accurate cycle counts with low variance and eliminates contamination from instructions executed after the code that we aim to measure. `rdtsc` and `rdtscp` store the current number of cycles in two registers. The high 32 bits of the cycle count are stored in the `edx` register, the low 32 bits are stored in the `eax` register. The upper 32 bits of the `rdx` and `rax` register are cleared [Int]. Using inline assembly we can invoke these instructions from C and shift the high values up. The implementation is shown in Listing 2. To easily reuse

```
static inline uint64_t tic() {
  uint64_t hi, lo;
  __asm__ volatile (
      "cpuid\n\t"
      "rdtsc\n\t"
      "mov %%rdx, %0\n\t"
      "mov %%rax, %1\n\t"
      : "=r" (hi), "=r" (lo)
      :: "%rax", "%rbx", "%rcx", "%rdx");
  return (hi << 32) | lo;
}
```

Listing 2: C code using inline x86 assembly for obtaining the start time of a measurement.

the snippet we encapsulate it in an `inline` function. The function is given the `inline` specifier to avoid including the cycles from calling a function in the measurement [Inl]. The counterpart function is similar and shown in Listing 3.

4.1.2 Measurement Noise Reduction

When we measure the execution time of a piece of code on any system, several system components introduce noise, causing us to have to measure more times to obtain a stable estimate of the mean or median execution time. To reduce noise, several controls may be introduced that are shown in Table 4.1. Many of these measures are not strictly

```
static inline uint64_t toc() {
  uint64_t hi, lo;
  __asm__ volatile(
      "rdtscp\n\t"
      "mov %%rdx, %0\n\t"
      "mov %%rax, %1\n\t"
      "cpuid\n\t"
      : "=r" (hi), "=r" (lo)
      :: "%rax", "%rbx", "%rcx", "%rdx");
  return (hi << 32) | lo;
}
```

Listing 3: C code using inline x86 assembly for obtaining the end time of a measurement.

Control	Description
Flush cache	Use the `clflush` instruction to flush the ciphertext and secret key.
Random alignments	Select a random alignment of the ciphertext and secret key.
Process priority	Give the running process a niceness of -20. The OS scheduler will prefer to run the application whenever possible.
Pin the application to a dedicated core	Using `taskset` we can pin all other processes to a different core, and dedicate a single CPU core to the application performing the measurements.
Use dedicated hardware	The hardware should not be shared with other VMs.

Table 4.1: Controls that may be introduced to lower execution time measurement noise.

necessary as the timing variation exploited in this thesis is exploitable without them. However, the noise reduction incurred by these measures is noticeable and can drastically reduce the number of required decapsulation oracle calls. Unfortunately, flushing CPU caches increases time required to obtain a measurement dramatically. Therefore, we only perform non-costly measures such as CPU core pinning, dedicated hardware and process prioritization.

4.1.3 Timing Decapsulations

We generate a single key-pair and 100 valid ciphertexts using hqc-r6-bch-128. We then time the decapsulation algorithm using the `tic` and `toc` functions. For each ciphertext we run the decapsulation function $100 \cdot 10^3$ times.

Data Analysis

We perform Welch's t-test [Wel47] for every pair of ciphertexts to determine if there is a difference in the mean time that a ciphertext takes to decrypt. This already yields interesting results: many ciphertext pairs exhibit a statistically significant difference in decapsulation time. This may be observed in Fig. 4.1, where the p-values of Welch's t-test are shown for pairs of ciphertexts. Approximately 65% of ciphertext pairs emit a difference in runtime of the decapsulation function. We classify a ciphertext pair as emitting a runtime difference when the p-value of Welch's t-test is ≤ 0.05. The larger the number of samples we take, the smaller the variance of the t-distribution: the variance of the t-distribution is $\frac{\nu}{\nu-2}$ when $\nu > 2$ where ν is the degrees of freedom. As we take more samples the degrees of freedom increases, reducing the variance of the t-distribution. A lower variance in the t-distribution results in higher p-values for the same difference in mean runtime. If the mean runtime difference is much larger than the standard deviation of the two distributions, the p-value is very low. We observe such a case here: the p-value is very low (median $p \approx 1.9 \cdot 10^{-111}$) when a difference was detected ($p \leq 0.05$). Such a low p-value gives us great confidence that there is some timing variation. However, this does not necessarily imply that the implementation is vulnerable to timing attacks. We must investigate the implementation further to determine if the timing-variation is connected to secret information.

To zero in on the cause of this difference we enhance our setup to time each step of the decapsulation algorithm. We divide the decapsulation function into four different coarse parts: "Key loading", "Decryption", "Re-encryption" and "Shared secret". This division is arbitrary, and a different segmentation would work just as well. For each part we then perform Welch's t-test to detect a difference in computation time between different ciphertexts. In Fig. 4.2 we see the p-value for each ciphertext pair. We observe that the re-encryption step in Fig. 4.2c emits the same pattern as we observed in Fig. 4.1. The re-encryption is implemented by a call a function called `hqc_pke_encrypt`, which implements the encryption of messages using a given seed. The decryption and shared secret derivation are both not highlighted by this test. Note that in the current setup we do not detect the side-channel discovered by Guo, Johansson, and Nilsson [GJN20]. This is likely because we only consider valid ciphertexts. However, the testing fixture could be adapted to introduce bit errors into the ciphertext with relative ease. Additionally, the key loading step emits a statistically significant difference in computation time. We presume this may be due to caching or alignment effects, as this section only consists of a series of `memcpy` calls. Notably, the median of the absolute differences for ciphertext pairs where a difference in computation time was detected ($p < 0.05$) is 205 cycles for the key loading part. This difference is dwarfed by the 7736 cycles difference for the re-encryption. We thus investigate the re-encryption part further and dismiss the results for the key loading part.

We recurse into the `hqc_pke_encrypt` function, which implements the re-encryption, split up the function into four coarse parts again: "Init", "Sample vector", "Encode" and "Add error" and perform the same analysis again. In Fig. 4.3 the p-value plots are

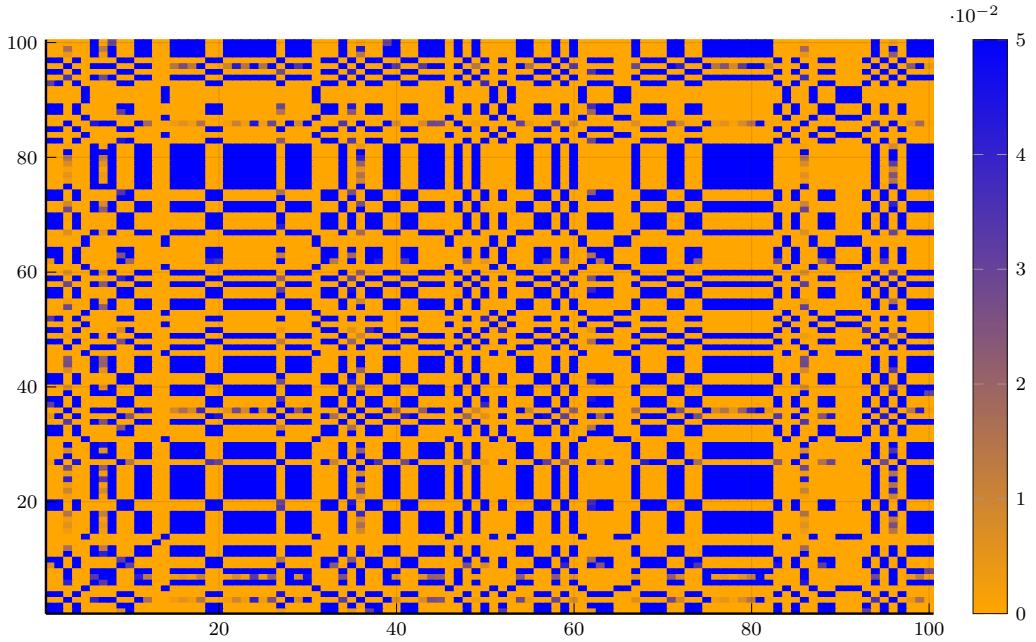


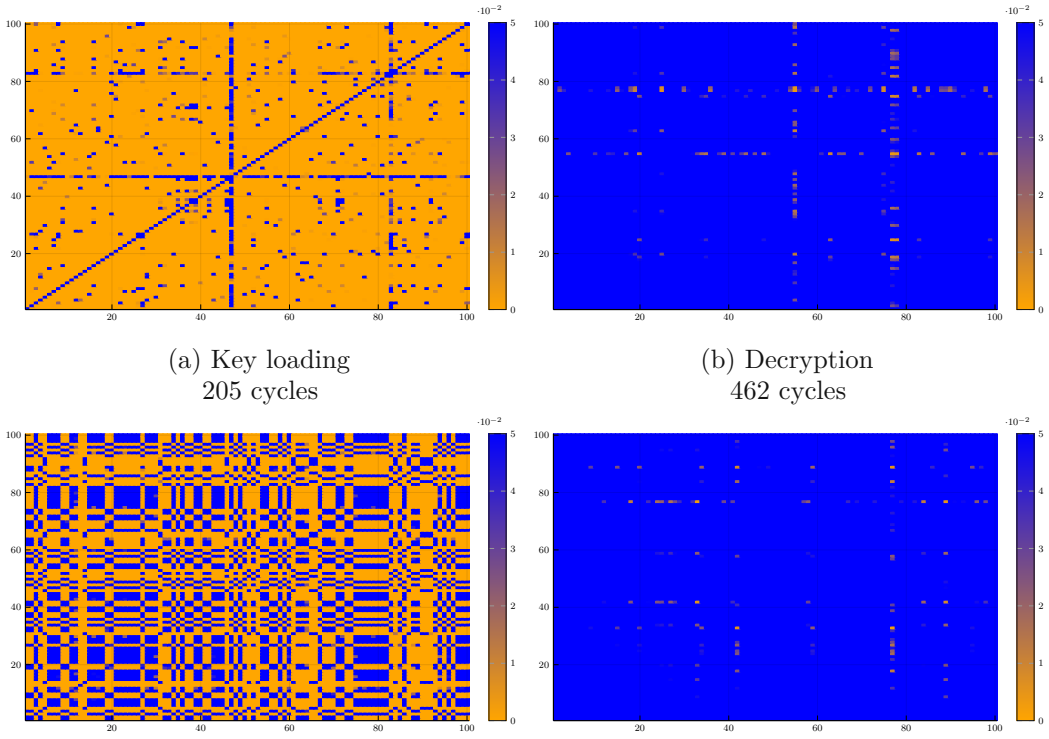
Figure 4.1: P-values of Welch's t-test for the decapsulation of 100 random ciphertexts. Each cell corresponds to a pair of ciphertexts $(\mathbf{c}_1, \mathbf{c}_2)$. Orange indicates that a statistically significant difference in decapsulation time was detected. Data was generated on hqc-r6-bch-128. Median absolute difference of statistically significant differences ($p < 0.05$): 8260 cycles ($\approx 4.13\,\mu s$ @ 2 GHz).

again shown for each part. This time, only the vector sampling part emits a statistically significant timing difference as seen in Fig. 4.3b. The pattern is again the same as in Fig. 4.2c and Fig. 4.1. We thus conclude, that the second part of the encryption function is responsible for the observed timing difference – it consists of three calls to `vect_set_random_fixed_weight`.

The complete source listings marking the concrete parts that the functions were divided into may be found in Section A.1. Note, however, that the segmentation may be chosen arbitrarily, since we would still be able to identify `vect_set_random_fixed_weight` as the cause of the timing variation.

Deterministic Rejection Sampling

HQC uses rejection sampling to obtain random vectors of a given weight w. This algorithm is non-constant time in the used seed and implemented by the `vect_set_random_fixed_weight` function. The function, shown in Listing 4, works in the following way: in each iteration a random bit position is sampled within the inner `do while` loop. If the bit position has already been sampled before, the sample is



(a) Key loading
205 cycles

(b) Decryption
462 cycles

(c) Re-encryption
7736 cycles

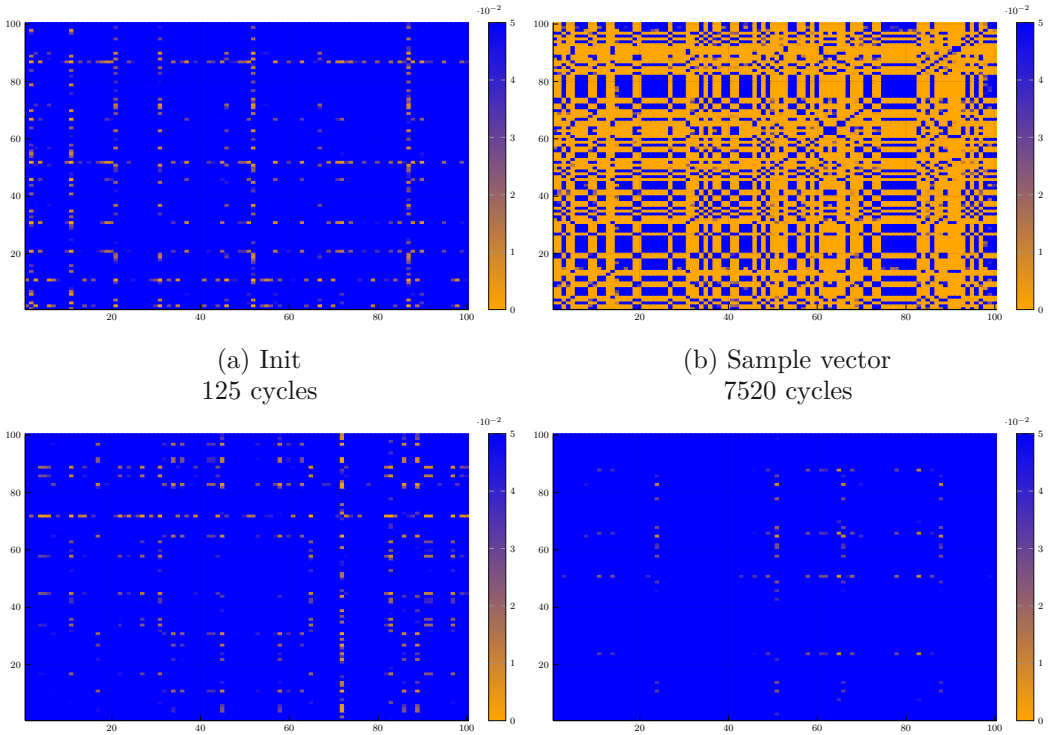
(d) Shared secret
92 cycles

Figure 4.2: P-values of Welch's t-test for 4 parts of the decapsulation function in hqc-r6-bch-128 and the median absolute timing difference in number of cycles. Orange indicates that a statistically significant difference was detected.

rejected. Otherwise, the bit position is stored in an array. The algorithm iterates until `weight` distinct bit positions have been sampled. At the end the vector is constructed by setting the bit positions that were sampled. The number of times a bit position collides with a previous bit position is directly proportional to the runtime of the algorithm. The randomness in the `vect_set_random_fixed_weight` function is deterministic and determined by an eXtendable-Output Function (XOF). An XOF can be used like a deterministic Random Number Generator (RNG) that uses a given seed to generate an arbitrary number of random bits. 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 $\theta = \texttt{SHA3}(m)$. The message $\mathbf{m}$ is obtained from the decryption of the underlying PKE scheme:

$$\mathbf{m} = \mathcal{C}.\mathrm{Decode}(\mathbf{v} - \mathbf{u} \cdot \mathbf{y}). \tag{4.1}$$

Therefore, the message $\mathbf{m}$ a ciphertext decrypts controls how many iterations the rejection



(a) Init
125 cycles

(b) Sample vector
7520 cycles

(c) Encode
59 cycles

(d) Add error
50 cycles

Figure 4.3: P-values of Welch's t-test for 4 parts of the encryption function in hqc-r6-bch-128 and the median absolute timing difference in number of cycles. Orange indicates that a statistically significant difference was detected.

sampling algorithm takes.

While it is likely hard to distinguish whether the rejection sampling runs for some n or $n+1$ iterations, there is a unique peculiarity within the implementation of HQC's rejection sampling algorithm that creates a large timing-variation: the `seedexpander` function is called a different number of times, depending upon the used seed.

Definition 4.1.1 (Additional `seedexpander` calls)**.** We refer to `seedexpander` calls which are executed conditionally – within the for loop – as *additional* `seedexpander` calls. In general, unless otherwise specified, we only count the number of additional `seedexpander` calls. In particular, this excludes `seedexpander` calls from the count that are always performed during a call to `vect_set_random_fixed_weight`.

The XOF 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

```
1  void vect_set_random_fixed_weight(
2   AES_XOF_struct *ctx, uint64_t *v, uint16_t weight) {
3
4   size_t random_bytes_size = 3 * weight;
5   uint8_t rand_bytes[3 * PARAM_OMEGA_R] = {0};
6   uint32_t random_data = 0;
7   uint32_t tmp[PARAM_OMEGA_R] = {0};
8   uint8_t exist = 0;
9   size_t j = 0;
10
11  // [...]
12
13  seedexpander(ctx, rand_bytes, random_bytes_size);
14  for (uint32_t i = 0 ; i < weight ; ++i) {
15   exist = 0;
16   do {
17    if (j == random_bytes_size) {
18     // This call is only performed when the randomness has been exhausted
19     seedexpander(ctx, rand_bytes, random_bytes_size);
20     j = 0;
21    }
22    random_data = ((uint32_t) rand_bytes[j++]) << 16;
23    random_data |= ((uint32_t) rand_bytes[j++]) << 8;
24    random_data |= rand_bytes[j++];
25   } while (random_data >= UTILS_REJECTION_THRESHOLD);
26   random_data = random_data % PARAM_N;
27   for (uint32_t k = 0 ; k < i ; k++) {
28    if (tmp[k] == random_data) {
29     exist = 1;
30    }
31   }
32   if (exist == 1) {
33    i--;
34   } else {
35    tmp[i] = random_data;
36   }
37  }
38  // [...]
```

Listing 4: Annotated implementation of the rejection sampling algorithm in HQC from the updated (2020-10-01) round 3 submission package [Car+20].

will need to produce additional randomness by issuing another `seedexpander` call. The bit positions we sample 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 number from $\{0, \ldots, 2^{24}-1\}$ that is to be reduced modulo n, where $n < 2^{24}$. However, the number is rejected if it is above the

largest multiple of n that is smaller than 2^{24}: $\eta := \left\lfloor \frac{2^{24}}{n} \right\rfloor n$. This is to avoid biasing the distribution, and discussed in detail in Section 7.1.1. In the HQC implementation η is called `UTILS_REJECTION_THRESHOLD`. Sampling distinct bit positions can thus fail in two ways:

1. The sampled number in $\{0, \ldots, 2^{24} - 1\}$ is larger than η
2. The sampled bit position collides with a previously sampled one

We can model rejection sampling of the number as a Bernoulli variable with success probability $p = \frac{\eta}{2^{24}}$. Each attempt at generating a valid bit position less than 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 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 to approximately 82% for the hqc-r7-rmrs-128 parameters. The probabilities for the other parameter sets are listed in Table 4.2. Thus, for hqc-r7-rmrs-128, only $1 - \tilde{p} \approx 18\%$ of seeds θ result in at least one call to the `seedexpander` function within the for loop of the rejection sampling routine. The seed expansion step takes several hundred cycles and, using timing information, we can determine whether it is run.

Traversing the Callstack

We now show how this timing variation connects to the decapsulation routine. The relevant data flow is illustrated in Fig. 4.4. The message $\mathbf{m}$ the recipient decrypts influences the outcome of the `seedexpander` function. The output of the `seedexpander` function determines the number of times it is called. We first take a high level look at the decapsulation routine in Listing 5. In line 2 the ciphertext consisting of $(\mathbf{u}, \mathbf{v})$ is decrypted using the secret key `sk`. The resulting message is placed into `m`. Then, the seed θ is computed. It is used in the re-encryption of `m`. Viewing the snippet of the encryption function in Listing 6 we observe three calls to the previously discussed `vect_set_random_fixed_weight` function. Once for each of the random vectors: $\mathbf{r_1}, \mathbf{r_2}, \mathbf{e}$, where the weight parameters `PARAM_OMEGA_R` and `PARAM_OMEGA_E` are equal. Each of these calls is using the same seed expander, whose randomness depends upon the seed θ which was derived from the decrypted message `m`. In each of these three invocations there is a $1 - \tilde{p}$ chance that the `seedexpander` function is called at least once within the for loop. Thus, $(1 - \tilde{p})^3 \cdot 100\%$ of messages result in three or more

Table 4.2: HQC parameters, the success probability p of generating a bit position in the required range, the probability $\tilde{p}$ of completing the rejection sampling routine without exhausting the initially generated randomness, and the probability of a message causing at least 3 additional `seedexpander` invocations.

Alias	$n_1 n_2$	n	$\omega_e = \omega_r$	p	$\tilde{p}$	$(1-\tilde{p})^3$
hqc-r6-bch-128	23,746	23,869	77	$\frac{16756038}{2^{24}}$	$\approx 80.3\%$	$\approx 0.8\%$
hqc-r6-bch-192	45,194	45,197	117	$\frac{16768087}{2^{24}}$	$\approx 80.7\%$	$\approx 0.7\%$
hqc-r6-bch-256	69,252	69,259	153	$\frac{16760678}{2^{24}}$	$\approx 72.7\%$	$\approx 2.0\%$
hqc-r6-rmrs-128	20,480	20,553	77	$\frac{16775461}{2^{24}}$	$\approx 86.0\%$	$\approx 0.3\%$
hqc-r6-rmrs-192	38,912	38,923	117	$\frac{16775813}{2^{24}}$	$\approx 83.2\%$	$\approx 0.5\%$
hqc-r6-rmrs-256	59,904	59,957	153	$\frac{16728003}{2^{24}}$	$\approx 52.5\%$	$\approx 10.7\%$
hqc-r7-rmrs-128	17,664	17,669	75	$\frac{16767881}{2^{24}}$	$\approx 82.0\%$	$\approx 0.6\%$
hqc-r7-rmrs-192	35,840	35,851	114	$\frac{16742417}{2^{24}}$	$\approx 65.9\%$	$\approx 4.0\%$
hqc-r7-rmrs-256	57,600	57,637	149	$\frac{16772367}{2^{24}}$	$\approx 79.1\%$	$\approx 0.9\%$

```
1  // Decrypting
2  hqc_pke_decrypt(m, u, v, sk);
3
4  // Computing theta
5  sha3_512(theta, (uint8_t*) m, VEC_K_SIZE_BYTES);
6
7  // Encrypting m'
8  hqc_pke_encrypt(u2, v2, m, theta, pk);
```

Listing 5: Snippet from `crypto_kem_dec`

calls to `seedexpander`. If we start with such a ciphertext, there is a high probability that ciphertexts that decrypt to a different message will call `seedexpander` a different number of times and will consequently emit a different timing behavior during decryption. Note the abuse of notation: technically we cannot "decrypt" the ciphertext of a KEM, we can only decapsulate it. However the HQC KEM is built using the PKE scheme which does have a decryption function. When we say that we decrypt a ciphertext generated by the Encaps function of the HQC KEM we refer to using the decryption function as in the first step of the decapsulation function in Fig. 3.2.

Distinguisher

We want to distinguish whether a ciphertext $\mathbf{c}_1$ decrypts to the same message $\mathbf{m}$ as another ciphertext $\mathbf{c}_2$ based on the timing behavior (Definition 4.1.2) of each of them.

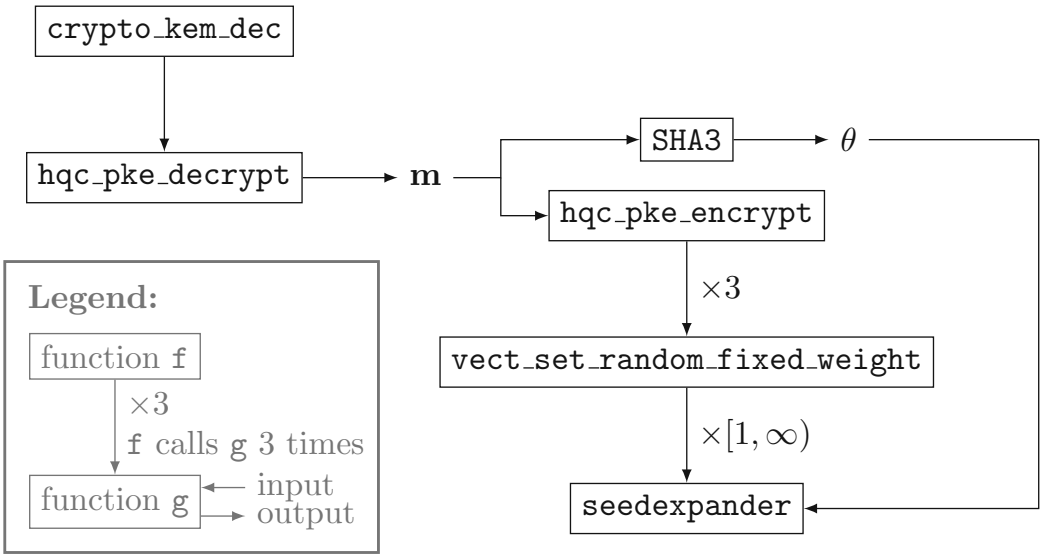


Figure 4.4: Information flow in the decapsulation function of the HQC KEM leading to the `seedexpander` function.

```
1  // Create seed_expander from theta
2  seedexpander_init(&seedexpander, theta, theta + 32,
3                    SEEDEXPANDER_MAX_LENGTH);
4
5  // Generate r1, r2 and e
6  vect_set_random_fixed_weight(&seedexpander, r1, PARAM_OMEGA_R);
7  vect_set_random_fixed_weight(&seedexpander, r2, PARAM_OMEGA_R);
8  vect_set_random_fixed_weight(&seedexpander, e, PARAM_OMEGA_E);
```

Listing 6: Snippet from `hqc_pke_encrypt`

Definition 4.1.2 (Timing Behavior of Ciphertext and Message)**.** The timing behavior of a ciphertext is the runtime required to decapsulate the ciphertext.

Since the timing behavior of a ciphertext depends on the message that the ciphertext decrypts to we define the timing behavior of a message $\mathbf{m}$ as the runtime of the `vect_set_random_fixed_weight` function calls inside the re-encryption using the seed θ derived from $\mathbf{m}$.

Given a fixed ciphertext $\mathbf{c}_1$ and the message $\mathbf{m}_1$ that it decrypts to, we want to identify whether another ciphertext $\mathbf{c}_2$ decrypts to the same message. We define a distinguisher $\mathcal{D}$ that returns 1 or true, if and only if the timing behavior of the ciphertexts is the same:

$$\mathcal{D}^{\mathcal{O}}(\mathbf{c}_1, \mathbf{c}_2) = \mathcal{O}(\mathbf{c}_1) \stackrel{?}{=} \mathcal{O}(\mathbf{c}_2) \tag{4.2}$$

where the oracle $\mathcal{O}(\mathbf{c})$ reveals the timing behavior $TB(\mathsf{sk}, \mathbf{c})$ of $\mathbf{c}$ under the secret key sk. We are interested in how well we can distinguish ciphertexts that decrypt to $\mathbf{m}_1$ from ciphertexts that do not. The advantage of this distinguisher is:

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

where $\mathcal{C}$ is the ciphertext space. This advantage reduces to:

$$\begin{aligned}|\Pr_{\mathbf{c}_2 \leftarrow\$ \mathcal{C}}[TB(\mathsf{sk}, \mathbf{c}_1) = TB(\mathsf{sk}, \mathbf{c}_2) \mid \text{Decrypt}(\mathsf{sk}, \mathbf{c}_1) = \text{Decrypt}(\mathsf{sk}, \mathbf{c}_2)] - \\ \Pr_{\mathbf{c}_2 \leftarrow\$ \mathcal{C}}[TB(\mathsf{sk}, \mathbf{c}_1) = TB(\mathsf{sk}, \mathbf{c}_2) \mid \text{Decrypt}(\mathsf{sk}, \mathbf{c}_1) \neq \text{Decrypt}(\mathsf{sk}, \mathbf{c}_2)]|\end{aligned}$$

Which we can further simplify to:

$$1 - \Pr_{\mathbf{c}_2 \leftarrow\$ \mathcal{C}}[TB(\mathsf{sk}, \mathbf{c}_1) = TB(\mathsf{sk}, \mathbf{c}_2) \mid \text{Decrypt}(\mathsf{sk}, \mathbf{c}_1) \neq \text{Decrypt}(\mathsf{sk}, \mathbf{c}_2)]$$

since ciphertexts that decrypt to the same message emit the same timing behavior. Therefore, the advantage depends on the probability that $\mathbf{c}_1$ and $\mathbf{c}_2$ have the same timing behavior, even though they do not decrypt to the same message. This depends on how rare the timing behavior of $\mathbf{c}_1$ is: if the timing behavior of $\mathbf{c}_1$ occurs with probability p_{tb}, the probability that another random ciphertext $\mathbf{c}_2$ has the same timing behavior, given that it does not decrypt to $\mathbf{m}_1$, is p_{tb}. Therefore, the advantage is $1 - p_{tb}$. Thus, the advantage is inversely proportional to the probability of the timing behavior of the message $\mathbf{m}_1$ that $\mathbf{c}_1$ decrypts to.

In our case more calls to the `seedexpander` function are less likely. Hence it is desirable to obtain a ciphertext that requires many `seedexpander` calls. However, a message that requires at least 4 additional `seedexpander` calls (Definition 4.1.1) does not exist with overwhelming probability: the chance of such a message existing is less than 2^{-183} for all parameters, as discussed in Section 6.3. We leave it to future research whether the structure of the hash function and the following XOF can be exploited to cause many bit position collisions. For example, in SHA2 such structure could be easy to find fixed-points of the underlying compression function due to its Davies-Meyer construction [MvV97, p.375]. Fixed points in the hash function could allow for more control over the seed θ

for the XOF. We are not aware of any such potential weaknesses in SHA3. For XOFs a NIST publication notes, that these primitives have "the potential for generating related outputs" [Dwo15, A.2]. We do not know whether any of these properties can be combined to control the flow in the `vect_set_random_fixed_weight` function. Assuming uniformity and independence of outputs for both SHA3 and the XOF we deem it unlikely that a message with 4 additional `seedexpander` calls exists let alone that it is feasible to efficiently find it.

Fortunately, such a message with 4 additional `seedexpander` calls is not a prerequisite for exploiting the timing variation. However, it does have an impact on the attack strategy used.

4.1.4 Timing Distribution

In the previous sections we have established that the message $\mathbf{m}$ that a ciphertext decrypts to influences the timing of the decapsulation function. We now investigate the timing distribution (Definition 4.1.3) for different messages.

Definition 4.1.3 (Timing Distribution)**.** A timing distribution of a function is a probability distribution over the amount of time that the function takes to execute. The time is given in CPU cycles.

More precisely, we can condition the timing distribution on the number of `seedexpander` calls a message results in. To that end we performed $10 \cdot 10^6$ decapsulations on a dedicated machine with no other load. Each decapsulation decapsulates a ciphertext generated honestly by the encapsulation function. Figure 4.5 shows the resulting probability density, which is the unconditioned density of the decapsulation function over the number of cycles that the function took to complete. The distribution clearly shows multiple modes. Vertical bars mark the modes of the empirical measurements conditioned on the number of `seedexpander` calls. The conditional probability density is shown in Fig. 4.6. Here we can clearly see that the number of `seedexpander` calls explains a the difference in timing very well.

4.2 Attacking HQC

Our attack works the following way: We pick a message $\mathbf{m}$ that has the property of resulting in 3 additional `seedexpander` calls (Definition 4.1.1). From Section 4.1.3 we know that most messages do not share this property with our chosen message $\mathbf{m}$. Therefore, if we can determine whether a decryption took exactly 3 calls or not, we can distinguish whether a ciphertext decrypts to the same message $\mathbf{m}$ with a high advantage, as demonstrated in Section 4.1.3.

Next, we generate a ciphertext $(\mathbf{u}, \mathbf{v})$ by manually setting $\mathbf{r_1}$ to $\mathbf{1}^n$, and $\mathbf{r_2}$ and $\mathbf{e}$ to $\mathbf{0}^n$ during the encryption of $\mathbf{m}$. This ciphertext has the desirable property, that its error is $\mathbf{y}$ – a part of the secret key:

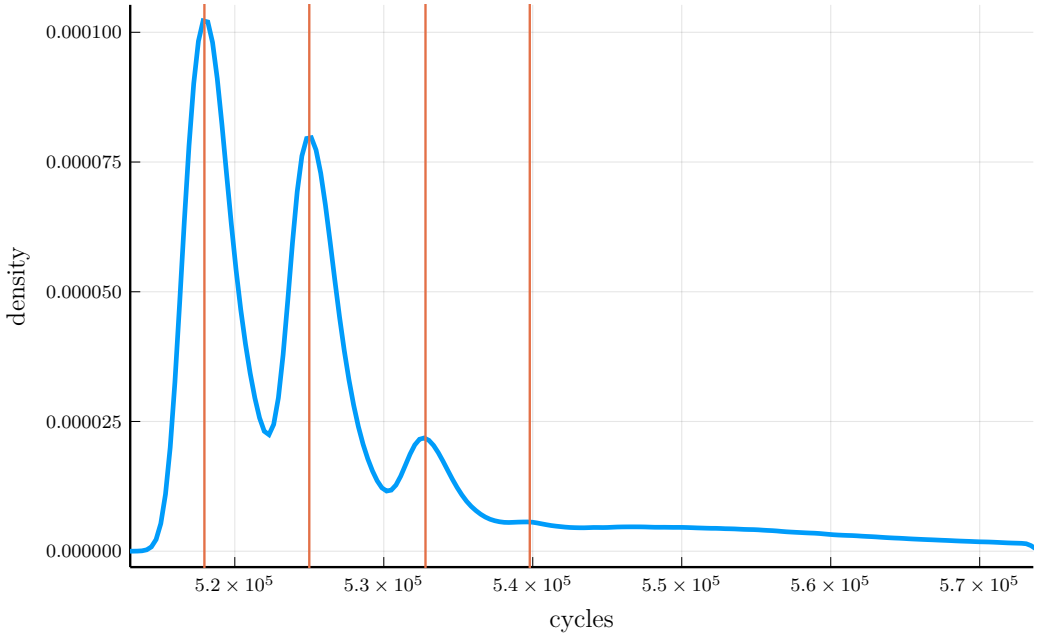


Figure 4.5: Probability density of the number of clock cycles required for the decapsulation function for random messages in hqc-r6-bch-128. The vertical bars show the median time for a given number of `seedexpander` calls.

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

If we can find out the error $-\mathbf{y} = \mathbf{y}$ that the decoder has to correct during the decryption, we can compute the remaining part of the secret key as $\mathbf{x} = \mathbf{s} - \mathbf{h} \cdot \mathbf{y}$[1]. 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 `vect_set_random_fixed_weight` function. Fortunately, 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. Now we wish to recover the error. Our approach follows the basic principles outlined by Hall, Goldberg, and Schneier [HGS99]. The authors propose adding an error e' to the ciphertext c until we detect that the modified

[1]Note that we do not need $\mathbf{x}$ as it is never used during decapsulation.

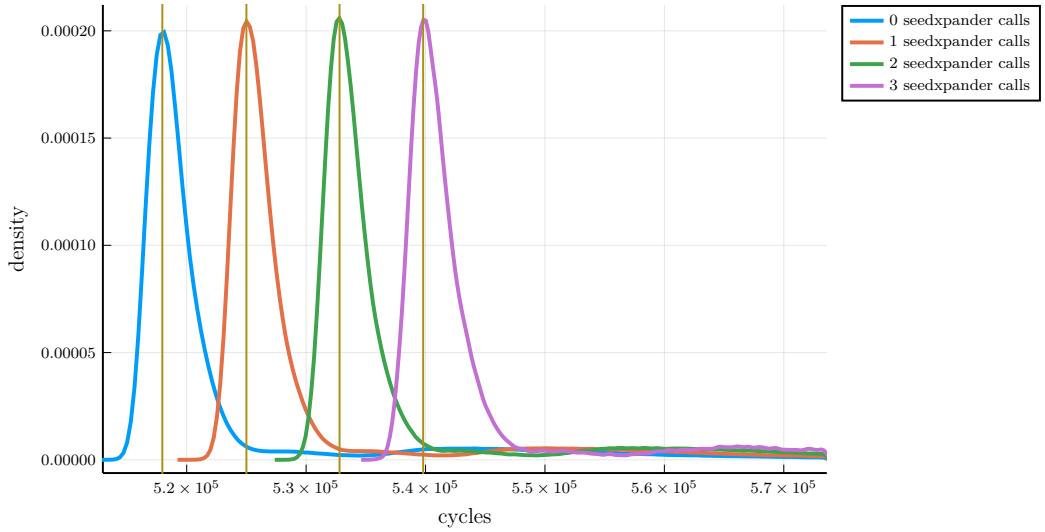


Figure 4.6: Conditional probability density of the number of clock cycles for the decapsulation function. Conditioned on the number of `seedexpander` calls. The vertical bars show the median time for a given number of `seedexpander` calls.

ciphertext c' decrypts to a different message 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 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 apply this method to HQC directly for several reasons:

1. We do not wish to correct the errors. We would like to determine the error e of our original ciphertext $c = mG + e$.
2. In more than 90% of cases, when flipping erroneous bits in the modified ciphertext the ciphertext does not decrypt back to the original message. Thus we would not detect that the bit is an error.
3. Our timing side-channel cannot 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.

We study the cause of the latter two issues in greater detail in Section 4.2.1.

The first issue can be solved with ease if we keep track of the error 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 m, we know that b is an error in $c' = c + e + e'$. Let $e'' = e + e'$. If the bit b is an error in e'', then b is an error in e if and only if the b-th bit of 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 increases the number of required oracle calls tremendously. 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 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.

Concretely, we can apply this method to HQC by adding an error $\mathbf{e}'$ to $\mathbf{v}$. During the decryption this adds our error $\mathbf{e}'$ to the error $\mathbf{y}$:

$$\begin{aligned}\text{Decrypt}(\text{sk}, (\mathbf{u}, \mathbf{v} + \mathbf{e}')) &= \mathcal{C}.\text{Decode}(\mathbf{v} + \mathbf{e}' - \mathbf{u} \cdot \mathbf{y}) \\ &= \mathcal{C}.\text{Decode}(\mathbf{mG} + \mathbf{e}' - \mathbf{y})\end{aligned}$$

Starting with an error of $\mathbf{e}' = 0$, we iteratively increase the weight of the error $\mathbf{e}'$ by flipping single bits in it. After each flip, we send the modified ciphertext $(\mathbf{u} + \mathbf{e}', \mathbf{v})$ to the decryption-timing oracle and check if the ciphertext takes a different amount of time to decrypt 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 b in $\mathbf{u} + \mathbf{e}'$, we flip the bit and send $(\mathbf{u} + \mathbf{e}' + 2^b, \mathbf{v})$ to the decryption-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 $\mathbf{c}$, we assume that it decrypted back to our original message $\mathbf{m}$. Otherwise, we assume that the ciphertext decrypted to a different message.

4.2.1 Attack in Detail

Start with ciphertext c that takes 3 `seedexpander` calls. We split the attack into three algorithms. Two algorithms attempt to recover the error $\mathbf{y}$, and a third one performs these algorithms multiple times to obtain multiple votes for a majority vote. Algorithm 4.1 and 4.2 follow the basic principle outlined by Hall, Goldberg, and Schneier [HGS99]. Algorithm 4.1 starts with our ciphertext c and finds a ciphertext c' that decrypts to a different message. In particular c' should have exactly one more error than the decoder could correct. From this the authors deduce that flipping any bit in c' and checking whether the ciphertext decodes again reveals whether that bit was an error bit in c or not. Unfortunately, as we will demonstrate later, this does not hold true in general, and in particular is not true for the ECCs used in HQC. Algorithm 4.2 then flips each bit and checks whether the new ciphertext decodes correctly or not. If it does, we deduce that the bit is an error bit and record this in e. Finally, Algorithm 4.3 performs Algorithm 4.1

and Algorithm 4.2 multiple times until sufficient votes have been gathered for all bits. The `until` loop condition in Eq. (4.3) checks whether there are sufficient votes for all bits.

$$\forall b.\ r[b] \geq \left\lfloor \frac{N}{2} \right\rfloor + 1 \ \ \vee \ \ c[b] - r[b] \geq \left\lfloor \frac{N}{2} \right\rfloor + 1 \tag{4.3}$$

The c array records the total number of votes that have been cast for each bit b. The r array records the number of 1-votes for each bit b, i.e., the number of votes that the bit b is set. To reach a majority either the number of 1-votes or the number of 0 votes has to exceed $\left\lfloor \frac{N}{2} \right\rfloor + 1$. The number of 0 votes for a bit b is calculated as $c[b] - r[b]$.

Algorithm 4.1: Find ciphertext $c' = (\mathbf{u}, \mathbf{v}')$ that decrypts to a different message

Input: c

Result: c', flipped bits bs

$c' \leftarrow c$
$bs \leftarrow$ random_permutation($\{1, \ldots, n_1 n_2\}$)
for $i \in bs$ **do**
  Flip bit $bs[i]$ in $\mathbf{v}$ of c'
  if $\neg \mathcal{D}^{\mathrm{TB}(\mathsf{sk},\cdot)}(c, c')$ **then**
    return $(c',\ bs[1, \ldots, i])$
  end
end

Algorithm 4.2: Recover combined error

Input: Original Ciphertext c and modified ciphertext c'

Result: Combined error $\mathbf{e}$

$\mathbf{e} \leftarrow \mathbf{0}$
for $i \in \{1, \ldots, n_1 n_2\}$ **do**
  Flip bit i in $\mathbf{v}$ of c'
  if $\mathcal{D}^{\mathrm{TB}(\mathsf{sk},\cdot)}(c, c')$ **then**
    Set bit i in $\mathbf{e}$
  end
  Flip bit i in $\mathbf{v}$ of c'
end
return $\mathbf{e}$

Algorithm 4.3: Recover target error

Input: ciphertext c and majority of N

Result: error e

Set $\forall b \in \{1, \ldots, n_1 n_2\}$. $c[b], r[b] \leftarrow 0$

repeat

  $(c', bs) \leftarrow$ *Algorithm* 4.1(c)

  $e' \leftarrow$ *Algorithm* 4.2(c, c')

  for $b \in \{1, ..., n_1 n_2\}$ **do**

    if *Bit* b *is set in* e' **then**

      $c[b] \leftarrow c[b] + 1$

      $r[b] \leftarrow r[b] + (b \notin bs)$ `// increment the counter if we did not introduce the error`

    end

  end

until $\forall b.\ r[b] \geq \left\lfloor \frac{N}{2} \right\rfloor + 1 \ \vee \ c[b] - r[b] \geq \left\lfloor \frac{N}{2} \right\rfloor + 1$

`// A majority is reached for all bits`

Set $\forall b \in \{1, \ldots, n_1 n_2\}$. $e[b] = r[b] \geq \left\lfloor \frac{N}{2} \right\rfloor + 1$

return e

False Positives and False Negatives

False positives arise strictly from the fact that our timing side-channel oracle sometimes classifies other messages as our original message m. If during the error recovery step, the bit we flip adds to the error in a way that results the message to decrypt to another message m' which has the same timing side-channel behavior as our chosen message m, we will detect false positives. We illustrate how false positives could arise using the example of a repetition code over $\mathbb{F}_2$:

$$G_{\mathrm{rep}_{2\times 3}} := \begin{pmatrix} 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 \end{pmatrix} \tag{4.4}$$

We encode the message $m = (01)$ as the codeword $c = mG_{\mathrm{rep}_{2\times 3}} = (000111)$. Now we iteratively flip bits in c until it decodes to a different message. This process is displayed in Table 4.3. As per our algorithm we find $c' = c + e$ in iteration 3 with the error vector $e = (011100)$. Now we would flip each bit in the word c' to determine if it is an error. Notice, that no false positives arise from the error classification seen in Table 4.4. However,

Iteration	Flipped Bit	Word	Decoded Message
0		(000|111)	(0|1)
1	2	(010|111)	(0|1)
2	4	(010|011)	(0|1)
3	3	(011|011)	(1|1)

Table 4.3: Toy example implementing Algorithm 4.1

Table 4.4: Classifications of bits during the error recovery step in Algorithm 4.2, starting with the original word $c' = (011|011)$. A bit is classified as an error when flipping it causes m' to equal $m = (0|1)$.

Flipped Bit	Word	Decoded m'	Classified as Error	Is Error	Result
1	(111|011)	(1|1)	no	no	true negative
2	(001|011)	(0|1)	yes	yes	true positive
3	(010|011)	(0|1)	yes	yes	true positive
4	(011|111)	(1|1)	no	yes	false negative
5	(011|001)	(1|0)	no	no	true negative
6	(011|010)	(1|0)	no	no	true negative

lets assume, that the message (0|1) has the same timing side-channel behavior as the message (1|0). This would imply, that we would classify bits 4 and 5 as errors, resulting in false positives. Additionally, we observe how false negatives can occur: flipping bit 3 does not cause the word to decode to a different message, since it is not part of the error relevant for the word to decode to a different message. This reveals a fundamental flaw with directly applying the attack by Hall, Goldberg, and Schneier [HGS99]. In their paper the authors make the assumption that we can determine a ciphertext with exactly $t+1$ errors, where t is the packing radius. However, as illustrated for the repetition code, this is not necessarily the case. Whether we do obtain such a ciphertext can depend upon the codeword and the order in which we introduce errors.

4.2.2 Success Probability Analysis

For each bit $i \in \{1, \ldots, n_1 n_2\}$ of the error, we collect bit guesses from the oracle $\mathcal{D}^{\mathrm{TB}(\mathsf{sk},\cdot)}(\mathbf{c}, \cdot)$ revealing whether a ciphertext decrypts to a certain plaintext until a majority of a total N samples is formed. This oracle is only correct with probability $p_c \approx 1 - (1 - \tilde{p})^3$, which can be computed from Table 4.2. We model the random variable X_i, the event that the final result for the i^{th} bit is correct, as iid Bernoulli variables. We model $\Pr[X_i = 1]$ as a Bernoulli process of iid Bernoulli variables Y_i with success probability p_c and N trials. We are interested in the probability of our guesses forming a majority:

$$\begin{aligned}\Pr[X_i = 1] &= \Pr[Y_i \geq \left\lfloor \frac{N}{2} \right\rfloor + 1] \\ &= \sum_{k \in \{\lfloor \frac{N}{2} \rfloor + 1, \ldots, N\}} \Pr[Y_i = k] \\ &= \sum_{k \in \{\lfloor \frac{N}{2} \rfloor + 1, \ldots, N\}} \binom{N}{k} p_c^k (1 - p_c)^{N-k}\end{aligned} \tag{4.5}$$

Then, assuming independence, the probability that all bits are correct is:

$$\prod_i \Pr[X_i = 1] = \Pr[X_1 = 1]^{n_1 n_2} \tag{4.6}$$

The resulting success probabilities for varying number of trials N are displayed in Fig. 4.7.

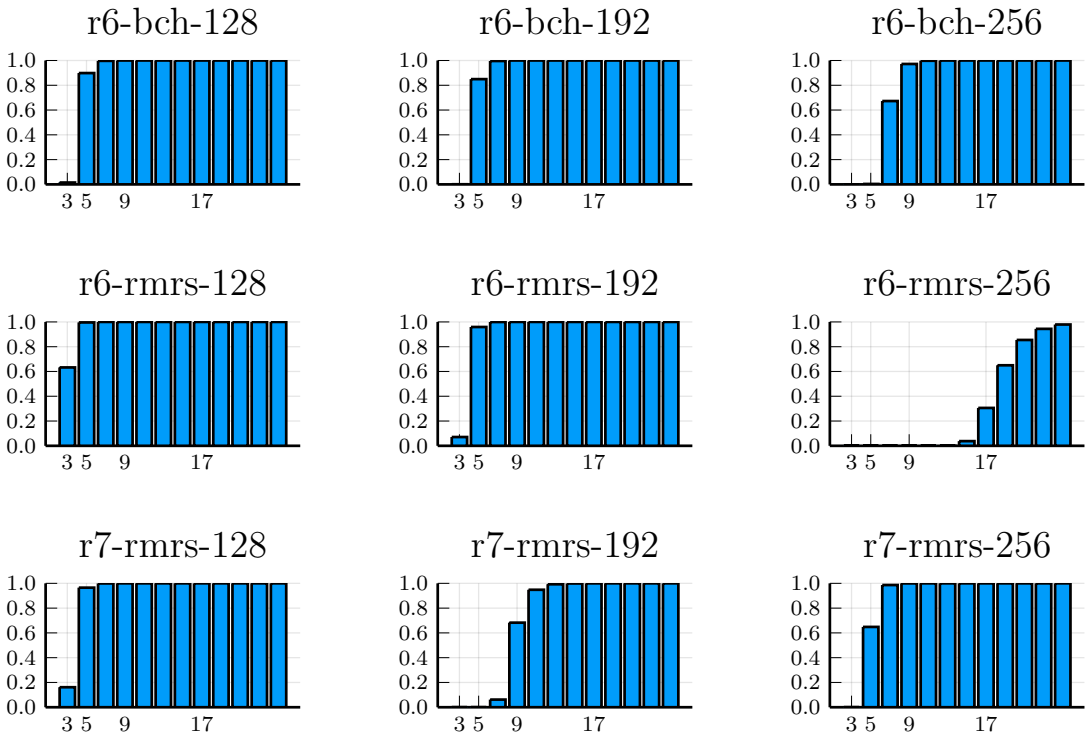


Figure 4.7: Success probabilities for each parameter set derived from the probability p_c. The X axis determines the number of samples that are drawn for each bit.

4.3 Optimized Attack

We now turn to a method inspired by an earlier timing side-channel attack on HQC discovered by Wafo-Tapa, Bettaieb, Bidoux, Gaborit, and Marcatel [WT+19]. The attack bases on the ability to distinguish whether the outer ECC used in HQC (either a BCH or RS code) corrects 0 or 1 errors. Unfortunately we do not have such an oracle, as we do not get timing side-channel on the outer decoder. However, we can forge such an oracle from the oracle that we have. The timing side-channel we have access to reveals whether a ciphertext decodes to the same message as another. The outer code can correct at least δ errors. The word of the outer code is an element of $\mathbb{F}_q^n$. Empirically we find that if we corrupt δ random coordinates of this word, corrupting any more coordinates will result in a decoding failure. The coordinates of the outer code are obtained by decoding words of the inner code. Therefore we obtain an oracle, whether the inner code decoded correctly or not. While this oracle is not directly the kind of oracle assumed in the [WT+19] attack, we can construct an analogous attack to obtain the errors in all but the δ coordinates which we corrupted. To obtain these remaining δ coordinates, we simply corrupt different δ coordinates, and perform the attack again.

The attack outlined in [WT+19] is sophisticated method for determining the error in a block for up to 2 errors. We propose a more general algorithm, following from a reformulation of the problem as a game. This allows us to apply algorithms such as minimax or expectiminimax, variants of which were employed by Donald Ervin Knuth to solve the "Mastermind" game, which is in many ways similar to the problem at hand [Don77]. In Mastermind an opponent picks a combination and gives the attacker an oracle which can be used to reveal information about the combination. The attacker wishes to use the least number of queries to the oracle to determine the combination. Mastermind can be mapped to the problem at hand: instead of a combination, the opponent picks an error e that is added to a codeword c. The oracle reveals for a given additional error e' whether $c + e + e'$ decodes to the same message as $c + e$. Knuth's algorithm selects those queries, which would rule out the most number of possible codes in the worst case. The advantage of this algorithm is that it easily generalizes to an arbitrary number of errors. Unfortunately, the algorithm also has poor asymptotic performance in the number of candidate errors.

The problem with Knuth's algorithm [Don77] for solving Mastermind is that it requires us to keep track of every remaining possible error. There are $n_e = \sum_{k=0}^{e_{\max}^{\text{blocks}}} \binom{n_2}{k}$ possible errors. If we keep track of every remaining possible error, we would need $n_e \log_2 n_e$ bits of memory in the first iteration. To save memory we can iterate through all n_e errors and check if the error is consistent with the recorded query answer pairs. In any case, the function becomes prohibitively time or memory intensive for large n_e. However, for HQC using BCH and repetition codes n_e is feasibly small: for hqc-r6-bch-128 $n_e = 4992$ with $e_{\max}^{\text{blocks}} = 3$. The cardinality of the query space is not an issue since we can only consider a small subset of queries. Considering only 3 queries per weight has been successful in practice. Additionally, we can discard many weights: Under the assumption

that the weight of the error is in $\{0, \ldots, e_{\max}^{\text{blocks}}\}$ the weight of the action must be in $\{h - e_{\max}^{\text{blocks}}, \ldots, h + e_{\max}^{\text{blocks}}\}$ where $h = \frac{n_2+1}{2}$. This is because after adding these two errors it must be possible that the decoding fails. If the weight of the two errors combined is $\geq h$ the decoding fails. Since a vector of weight $e_{\max}^{\text{blocks}}$ added to a vector of weight $h - e_{\max}^{\text{blocks}} - 1$ never has a weight $\geq h$ we can simply not consider vectors of weight $\leq h - e_{\max}^{\text{blocks}} - 1$, because we know the decoding outcome beforehand. An analogous argument can be made for the upper bound.

To determine the best action, we need to determine the value of an action. The simplest method is the following: for each possible outcome – either it decodes correctly, or it does not – we count the number of errors that would result in that outcome. Let n_c be the number of errors consistent with the "decodes correctly" outcome and $n_{\neg c}$ be the number of errors consistent with the "does not decode correctly" outcome. Then in the worst case, the action rules out $\min(n_c, n_{\neg c})$ errors. This is the worst-case value of the action. We then pick the action with the highest worst-case value. To remediate errors received from the timing-decapsulation oracle we again perform a majority vote. For hqc-r6-bch-128 a majority of 3 samples has empirically proven sufficient to eradicate almost all failures.

A single round of the algorithm where the number of candidates is reduced is shown in Fig. 4.8.

4.3.1 Success Probability Analysis

We consider the success probability when recovering $n_1 n_2$ bits of the n bit secret key $\mathbf{y}$. The success probability of this attack comes down to the choice of $e_{\max}^{\text{blocks}}$.

This has already been partially analyzed in [WT+19]. In their analysis they define A_i to be the event where $\mathbf{y}$ has exactly i blocks with two 1s and no block with more set bits. The success probability with $e_{\max}^{\text{blocks}} = 1$ is equal to $\Pr[A_0]$: we choose ω out of n_1 blocks to contain a single 1 bit. These blocks have $\binom{n_2}{1} = n_2$ ways to position the single bit and we have have to choose a way ω times – once for each block that contains a 1. Thus $\Pr[A_0]$ can be computed as [WT+19]:

$$\Pr[A_0] = \frac{\binom{n_1}{\omega}\binom{n_2}{1}^{\omega}}{\binom{n}{\omega}} \tag{4.7}$$

For $e_{\max}^{\text{blocks}} = 2$ we can compute the disjoint union of A_i as [WT+19]:

$$\sum_{i=0}^{\lfloor n_1/2 \rfloor} \Pr[A_i] = \binom{n}{\omega}^{-1} \sum_{i=0}^{\lfloor n_1/2 \rfloor} \binom{n_1}{i}\binom{n_2}{2}^{i}\binom{n_1 - i}{\omega - 2i}\binom{n_2}{1}^{\omega-2i} \tag{4.8}$$

For hqc-r6-bch-128, this evaluates to only $\approx 66\%$. The authors claim that for $e_{\max}^{\text{blocks}} = 3$ the success probability is above 99% for $n_1 = 796, n_2 = 31, n = 24{,}677, \omega = 67$, however

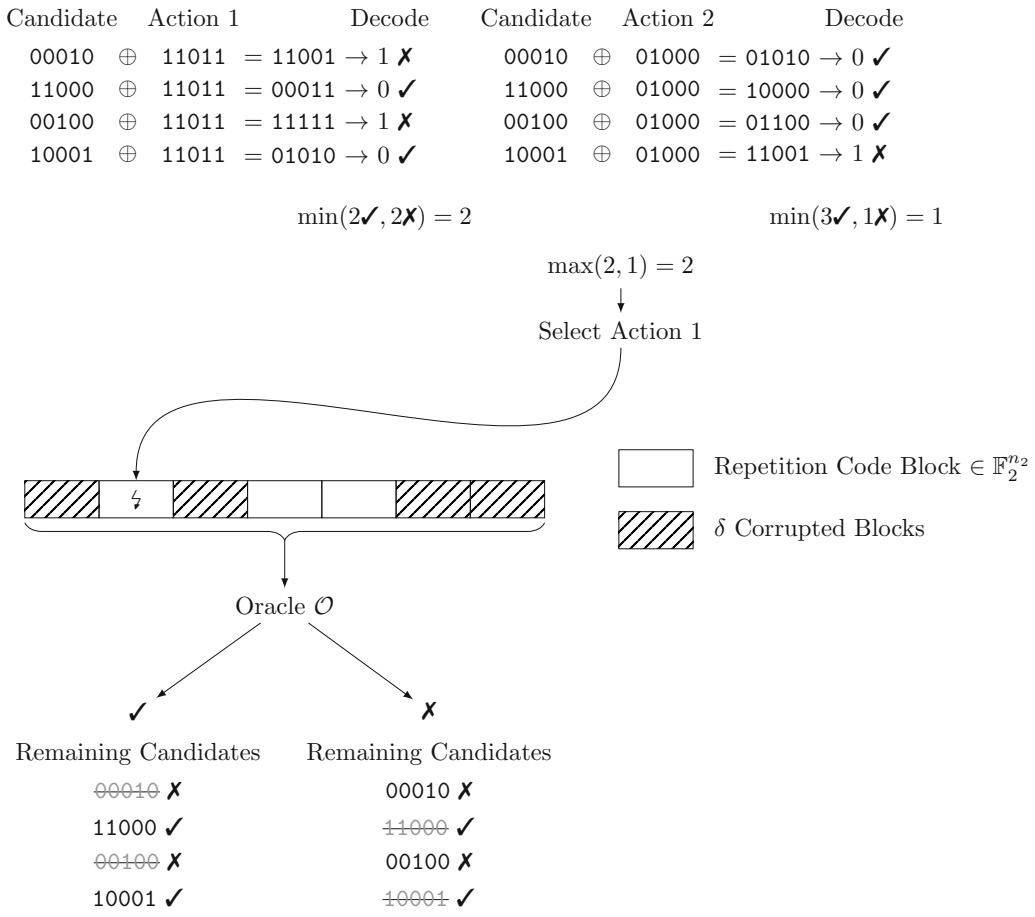


Figure 4.8: A single round of the optimized attack. We start with a set of remaining candidates for the error patterns. We then consider multiple actions and obtain how well they perform on reducing the number of candidates in the worst case (min). We then select the action that performs best in the worst case (max). The action is used to introduce additional errors into the ciphertext. In this case the 2nd block is faulted. The ciphertext is sent to the oracle and depending on whether it decoded or not the remaining candidates list is reduced in different ways. This core part is repeated for a block until there is only one error pattern left in the remaining candidates.

```
function sim(n1, n2, n, w, max_e_blocks, max_e_remainder)
 n1n2 = n1*n2

 pos = sample(0:n-1, w, replace=false)

 in_blocks = (div(p, n2) for p in pos if p < n1n2)
 max_set_bits_in_blocks = maximum(values(countmap(in_blocks)))

 outside_blocks = count(true for p in pos if p >= n1n2)

 blocks_success = max_set_bits_in_blocks <= max_e_blocks
 outside_success = outside_blocks <= max_e_remainder
 blocks_success && outside_success
end
```

Listing 7: Simulation function in Julia which simulates sampling a random secret key. It then counts the number of bits that fall into each block and returns whether that number is less than a given number.

they do not detail their analysis. Additionally, the analysis does not consider the case where some of the remaining $n - n_1 n_2$ bits are non-zero. This case happens $\approx 29\%$ of the time for some parameter sets as shown in Section 4.4. For example, when a bit is set in the $n - n_1 n_2$ bits, we only have $\omega - 1$ blocks that contain a 1, assuming no block contains more than a single set bit.

To analyze the full situation we opt to perform a Monte-Carlo simulation instead of a formal analysis, as we only want to determine a $e_{\text{max}}^{\text{blocks}}$ and $e_{\text{max}}^{\text{remainder}}$ that is sufficient for high success rates. Compared to [WT+19] our analysis also considers cases where not all sampled bit positions fall into the $n_1 n_2$ bits, which makes a significant difference for some parameter sets. The `sim` function in Listing 7 samples the support of a random vector of weight w and checks whether all blocks contain $\leq e_{\text{max}}^{\text{blocks}}$ set bits and whether the number of set bits outside the $n_1 n_2$ bits is $\leq e_{\text{max}}^{\text{remainder}}$. We can then compute the empirical success probability for each parameter set for a given $e_{\text{max}}^{\text{blocks}}$ and $e_{\text{max}}^{\text{remainder}}$ as follows by sampling from the `sim` function many times as seen in Listing 8 and computing the mean. We show the simulation results in Table 4.5

We choose $e_{\text{max}}^{\text{blocks}} = 3$ and $e_{\text{max}}^{\text{remainder}} = 2$ for our attacks to maximize the chance of success while keeping the number of errors to consider sufficiently low. Note that our analysis does not consider incorrect information from the side-channel, therefore it may only function as an upper bound.

```
success = 0
count = 100000
for i in 1:count
 success += sim(params..., max_e_blocks, max_e_remainder)
end
success / count
```

Listing 8: Executing the simulation function many times to obtain an estimate of the expected value of its distribution.

$e_{\max}^{\text{blocks}}$ \ $e_{\max}^{\text{remainder}}$	0	1	2	3
1	$\approx$ 1.5%	$\approx$ 2.1%	$\approx$ 2.3%	$\approx$ 2.3%
2	$\approx$ 60.3%	$\approx$ 84.8%	$\approx$ 89.3%	$\approx$ 90.1%
3	$\approx$ 66.8%	$\approx$ 94.0%	$\approx$ 99.0%	$\approx$ 99.7%
4	$\approx$ 67.1%	$\approx$ 94.0%	$\approx$ 99.2%	$\approx$ 99.9%

(a) Success probability estimate for hqc-r6-bch-128

$e_{\max}^{\text{blocks}}$ \ $e_{\max}^{\text{remainder}}$	0	1	2	3
1	$\approx$ 0.0%	$\approx$ 0.0%	$\approx$ 0.0%	$\approx$ 0.0%
2	$\approx$ 67.6%	$\approx$ 68.0%	$\approx$ 67.9%	$\approx$ 68.0%
3	$\approx$ 97.9%	$\approx$ 98.7%	$\approx$ 98.7%	$\approx$ 98.7%
4	$\approx$ 99.2%	$\approx$ 100.0%	$\approx$ 100.0%	$\approx$ 100.0%

(b) Success probability estimate for hqc-r6-bch-192

$e_{\max}^{\text{blocks}}$ \ $e_{\max}^{\text{remainder}}$	0	1	2	3
1	$\approx$ 0.0%	$\approx$ 0.0%	$\approx$ 0.0%	$\approx$ 0.0%
2	$\approx$ 44.4%	$\approx$ 45.1%	$\approx$ 45.3%	$\approx$ 45.2%
3	$\approx$ 95.1%	$\approx$ 96.6%	$\approx$ 96.5%	$\approx$ 96.6%
4	$\approx$ 98.3%	$\approx$ 99.9%	$\approx$ 99.9%	$\approx$ 99.9%

(c) Success probability estimate for hqc-r6-bch-256

Table 4.5: Monte-Carlo simulation results for the r6-bch parameter sets.

4.3.2 Implementation

Code snippets for the following implementation details are shown in Section A.2. We start by obtaining a message that induces 3 `seedexpander` calls. Then, we perform up to `majority_of` iterations in which we gather votes on the bits of the secret key $\mathbf{y}$, which is the error in the decoder has to correct during decryption. In each iteration we perform the following: we corrupt δ positions in v. The `blocks_order` array contains a random shuffle of the numbers of the inner code blocks: $\{0, \ldots, n_1 - 1\}$. Now we recover the remaining non-faulted blocks. To recover a single block we generate all error patterns with weight $\{0, \ldots, e_{\text{max}}^{\text{blocks}}\}$. The `generate_error_patterns` function generates the support for all considered error patterns. The function stores its results in a struct called `Vecs`. `Vecs` is a custom dynamic array using amortized resizing. `Vec` is similar, except that it stores 8 bit integers that index into the inner codeword of size n_2. We now want to reduce the number of possible errors by querying the decapsulation oracle. To that end we loop until the number of possible error patterns (`error_patterns.len`) is 1. Inside the loop we sample 3 actions for each weight in $\{h - e_{\text{max}}^{\text{blocks}}, \ldots, h + e_{\text{max}}^{\text{blocks}}\}$. The `sample_actions` function uses rejection sampling to obtain patterns of a specific weight. Then we compute the action value for each action and select the best action. The value of an action is computed by the number of error patterns that it rules out in the worst case. We determine whether an action applied to a given error pattern decodes correctly, by checking whether the weight of the two combined is $\leq h - 1$, the maximum number of flips the repetition code can tolerate. The `combined_weight` function computes the weight of `action` $\oplus$ `error_pattern` using the exclusion inclusion principle. Once we have identified the best action of our sampled actions, we apply the action to v. We now query the idealized decryption timing oracle using the ciphertext $(\mathbf{u}, \mathbf{v})$, where $\mathbf{u} = \mathbf{1}^n$, and obtain whether the repetition code decoded correctly or not. Using this information we reduce the number of remaining possible errors. Here, `same_ti` is the information deduced from the timing decapsulation oracle: whether the repetition code decoded correctly or not. It signifies whether the ciphertext decrypted to $\mathbf{m}$ or to a different message, which is equivalent to whether the repetition code decoded correctly or not, except when the side-channel yields a false-positive. The `update_error_patterns` function removes error patterns that are inconsistent with the result obtained from the decapsulation oracle: we check for each error whether *error* $\oplus$ *action* decodes, and if it yields the same result as we obtained from the decryption timing oracle. If it does not, we remove the error from our list of possible `error_patterns`. We have now reduced the number of possible errors for the current inner-code block of interest. To end this iteration, we reset v to its original state. Once we have eliminated all except one error pattern, we have obtained a solution for the current block. We increment the respective counters, that keep track of the results for each bit. The `results` array keeps track of how many 1-votes there are for each bit, the `counters` array keeps track of how many votes have been cast. To recover the remaining blocks that were faulted, we set the start indices anew such that different blocks are faulted and we only recover the remaining previously faulted blocks. Then we execute the same procedure of recovering each block again. Finally, once sufficient votes have been recorded for every bit, we set the bits

according to the votes we recorded. As an optimization, we can only perform the block recovery procedure if we have not yet reached a majority decision for every bit in that block. This saves approximately half the decapsulation oracle calls as false-positives are relatively rare for the side-channel in hqc-r6-bch-128.

4.3.3 Attacking the RMRS Version

The attack we just described unfortunately only works for the version of HQC that uses a repetition code. When we try to extend the attack to the RMRS version we run into a dimensionality problem: codewords in the RM code are 128 bits long, and each codeword is repeated μ times, resulting in a total length of at least 384 bits per duplicated RM codeword. We would like to avoid having to either iterate through all n_e errors or having to store the remaining errors. An idea to achieve this, is to use an Satisfiability Modulo Theories (SMT) solver to obtain errors consistent with the queries and responses recorded so far. "Fastermind" [CK18] uses this approach to generate valid codes. The problem of finding such errors has been proven NP-complete for mastermind [SZ05]. Obtaining a few still viable errors is sufficient, to evaluate the value of a query. While this approach may be feasible, a more efficient approach would take the code structure of the RM code into account.

HQC RMRS encodes each symbol in $\mathbb{F}_{2^8}$ of the RS code using an $R(r = 1, m = 7)$ code. Each symbol in $\mathbb{F}_{2^8}$ of the RS code is mapped to multiple symbols in ${\mathbb{F}_2}^8$ and then encoded using the RM code. The RM code has a dimension of 8 and a codeword length of $2^7 = 128$. Then, each codeword of the RM code is repeated μ times, resulting in a parameter for $n_2 = 128 \cdot \mu$. In Listing 9 we can see the concatenated code for the RMRS code version of HQC. Decoding these concatenated codes works in the opposite direction, with the exception that the duplicated RM code is decoded using the Hadamard transform discussed in Section 2.2.3. Note that the repetition is not decoded using a repetition code decoder, but is decoded directly using the Hadamard transform decoder. Given a word $\mathbf{w} = (\mathbf{w}_1, \mathbf{w}_2, \ldots, \mathbf{w}_\mu) \in (\mathbb{F}_2^n)^\mu$ of the duplicated RM code the decoder computes the vector $\mathbf{s}$:

$$\mathbf{s} = \sum_{1 \leq j \leq \mu} (-1)^{\mathbf{w}_j} \tag{4.9}$$

where $\mathbf{s} \in \{-3, -1, 1, 3\}^{128}$ when $\mu = 3$. This reduces the replicated words down to a vector of the length of a single RM codeword. Then, the Hadamard transform is applied, yielding the correlations of the codeword with every walsh basis function:

$$\hat{\mathbf{s}} = \mathbf{H}_7 \mathbf{s} \tag{4.10}$$

and we select the coordinate j of maximum magnitude:

$$j = \arg\max |\hat{\mathbf{s}}| \tag{4.11}$$

We write out j in binary to obtain the message $\mathbf{m}$. If the amplitude of the correlations with the walsh function corresponding to the message $\hat{\mathbf{s}}_j$ is negative, we flip all bits in $\mathbf{m}$. The decoding algorithm is summarized in Algorithm 4.4.

Algorithm 4.4: RMDecode

Input: Word $\mathbf{w} \in (\mathbb{F}_2^{2^7})^{\mu}$

Result: Decoded message

1 $\hat{\mathbf{s}} = H \sum_{1 \leq j \leq \mu} (-1)^{\mathbf{w}_j}$

2 $pos = \arg\max |\hat{\mathbf{s}}|$

3 $bin\_pos = \text{to\_binary}(pos) \in \mathbb{F}_2^{2^7}$

4 **return** $\text{sgn}(\hat{\mathbf{s}}_{pos}) \cdot bin\_pos$

Recall that decryption works the following way:

$$\mathbf{m} = \mathcal{C}.\text{Decode}(\mathbf{v} - \mathbf{u} \cdot \mathbf{y}). \tag{4.12}$$

Here, $\mathbf{v}$ and $\mathbf{u}$ – both from $\mathbb{F}_2^n \cong \mathbb{F}_2[x]/\langle x^n - 1\rangle$ – are selected by the adversary. For our attacks so far it has proven useful to set $\mathbf{u}$ to 1 since we then directly obtain the secret key $\mathbf{y}$ from the error added to the word $\mathbf{v}$ which we supply. We can again perform the same trick as before, and corrupt δ coordinates of the outer RS code, to obtain an oracle whether the RM code decoded correctly. We segment $\mathbf{v}$ into n_1 blocks of $n_2 = 128 \cdot \mu$ bits:

$$\begin{aligned} \mathbf{v} = ((&\mathbf{v}_1^1, \mathbf{v}_2^1, \ldots, \mathbf{v}_\mu^1), \\ &(\mathbf{v}_1^2, \mathbf{v}_2^2, \ldots, \mathbf{v}_\mu^2), \\ &\vdots \\ &(\mathbf{v}_1^{n_1}, \mathbf{v}_2^{n_1}, \ldots, \mathbf{v}_\mu^{n_1})) \end{aligned} \tag{4.13}$$

We now look at a single word $\mathbf{w}$ in the duplicated RM code, for which we want to obtain the error. We omit the superscript $\cdot^i$ where the block number is not relevant. The decryption algorithm introduces errors $(\mathbf{e}_1, \ldots, \mathbf{e}_\mu)$ to the word that are components of the secret key $\mathbf{y}$. The components of the secret word $\mathbf{w} = (\mathbf{w}_1, \mathbf{w}_2, \ldots, \mathbf{w}_\mu)$ are:

$$(\mathbf{w}_1, \mathbf{w}_2, \ldots, \mathbf{w}_\mu) = (\mathbf{v}_1 + \mathbf{e}_1, \mathbf{v}_2 + \mathbf{e}_2, \ldots, \mathbf{v}_\mu + \mathbf{e}_\mu). \tag{4.14}$$

We are faced with the task of selecting $\mathbf{v}_i$ such that decoding failures reveal information about the errors $(\mathbf{e}_j)_{1 \leq j \leq \mu}$. A decoding failure reveals to us that either the position of the largest magnitude or the sign of the amplitude of the largest magnitude in $\hat{\mathbf{s}}$ changed. Thus, we obtain information on the spectrum of $\mathbf{s}$.

Table 4.6: Mapping from individual bits of $\mathbf{w}$ to $\mathbf{s}$.

$\mathbf{w}_1$	$\mathbf{w}_2$	$\mathbf{w}_3$	$(-1)^{\mathbf{w}_1}$	$(-1)^{\mathbf{w}_2}$	$(-1)^{\mathbf{w}_3}$	$\mathbf{s}$
0	0	0	1	1	1	3
0	0	1	1	1	−1	1
0	1	0	1	−1	1	1
0	1	1	1	−1	−1	−1
1	0	0	−1	1	1	1
1	0	1	−1	1	−1	−1
1	1	0	−1	−1	1	−1
1	1	1	−1	−1	−1	−3

Algorithm 4.5: DecodeWithError: decodes the given word $\mathbf{w}$ after adding a secret error $\mathbf{e}$

Input: Word $\mathbf{w}' \in (\mathbb{F}_2^{2^7})^{\mu}$

Result: Decoded message

1 **return** RMDecode($\mathbf{w}' + \mathbf{e}$)

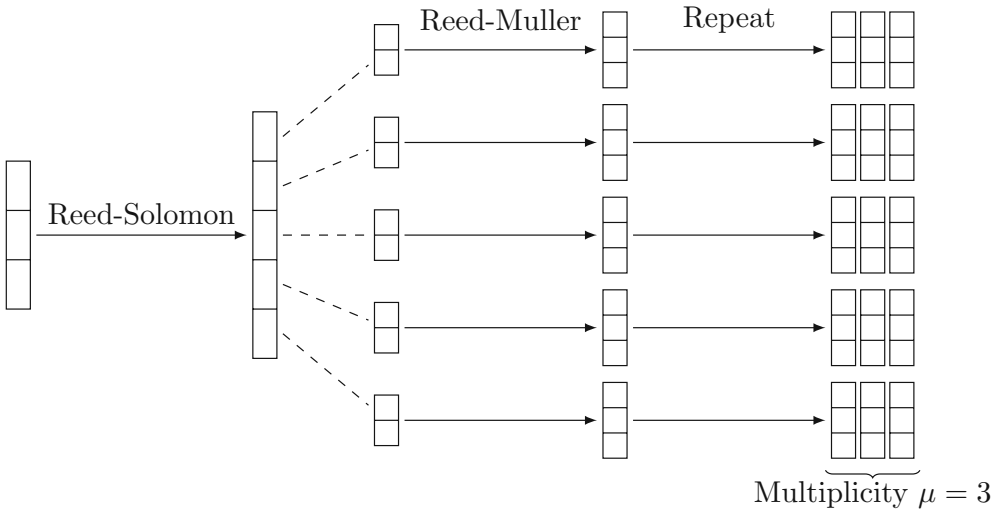
Sending a ciphertext to the decapsulation oracle, the ciphertext will take a different amount of time to decrypt depending on whether the RM code decoded correctly. The decapsulation oracle computes the function in Algorithm 4.5 using the attacker chosen $\mathbf{w}'$. The decryption time reveals whether the following equality holds:

$$\texttt{DecodeWithError}(\mathbf{c}) \stackrel{?}{=} \texttt{DecodeWithError}(\mathbf{c} + \mathbf{e}') \tag{4.15}$$

where $\mathbf{c}$ is a reference codeword and $\mathbf{e}'$ an additional error, both chosen by the adversary. We can choose $\mathbf{c}$ by changing the message.

Suppose we gained full knowledge of the spectrum $\hat{\mathbf{s}}$ of $\mathbf{s}$, we could invert it to obtain $\mathbf{s}$. However we still need to decompose $\mathbf{s}$ into the individual summands $(\mathbf{w}_j)_{1 \le j \le \mu}$ that form $\mathbf{s}$. The mapping $(\mathbf{v}, \mathbf{e}) \mapsto \mathbf{s}$ is not injective, therefore we need additional information. For $\mu = 3$ when a coordinate in $\mathbf{s}$ is either 3 or -3 we know that all corresponding coordinates in $(\mathbf{w}_j)_{1 \le j \le \mu}$ are 0 or 1, respectively. However, when a coordinate in $\mathbf{s}$ is 1 or -1, one of the summands is unlike the others, and we do not know which. This mapping is displayed in Table 4.6. One way to obtain additional information would be to recover the spectrum of $\mathbf{s}$ for varying $\mathbf{w}$. For example, we could obtain $\mathbf{s}$ for each of these modifications of $\mathbf{w}$: $\mathbf{w} + (0, 0, 1)$, $\mathbf{w} + (0, 1, 0)$, and finally $\mathbf{w} + (1, 0, 0)$. The $\mathbf{s}$ obtained for each uniquely identify the $\mathbf{s}$.

Lastly, if the spectrum has multiple positions with a maximum magnitude, the decoding algorithm could choose to decode to any of these corresponding messages. This presents an additional challenge as we have to account for the specific way in which the HQC implementation favors one codeword over another. This is non-trivial, since the SIMD



Listing 9: Concatenated Code in the RMRS version of HQC. The multiplicity μ is 3 for hqc-r7-rmrs-128.

implementation uses a binary search algorithm to determine the value of the maximum magnitude, and then identifies the position of a maximum magnitude using further scans across the spectrum. It is not immediately obvious whether this implementation follows the behavior of the reference implementation for all possible spectra.

We rest this case, and leave it to future work to determine an algorithm that can recover the secret error from the oracle described above. We could not transform the optimized attack to perform on the RMRS version, as we did not manage to find an efficient algorithm for determining the error of an RM codeword given the oracle that we are provided. However, the non-optimized version can break this version too. Therefore, the code choice does not protect against this side-channel attack.

4.4 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. 4.9 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 by Wafo-Tapa, Bettaieb, Bidoux, Gaborit, and Marcatel in their originally published paper. We have contacted the authors regarding the issue,

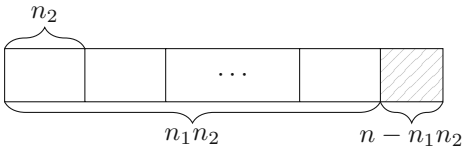


Figure 4.9: An element of $\mathbb{F}_2[x]/\langle x^n - 1\rangle$ and its segmentation into codewords of the inner code.

and they have acknowledged it. The authors stated they will add a remark bringing attention to this issue in the ePrint version of their publication [WT+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 123 bits in hqc-r6-bch-128. 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 only $\approx 70.7\%$ for hqc-r6-bch-128 when we guess that all remaining bits are zero, represented by $\Pr[Z = 0]$ in Table 4.7. 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. This also suggests picking $e_{\max}^{\text{remainder}} = 2$ for all parameter sets. Fig. 4.10 shows the probability distribution for each parameter set.

Table 4.7: 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.

Alias	$n_1 n_2$	n	ω	$n - n_1 n_2$	$\sum_{i=0}^{2} \binom{n-n_1 n_2}{i}$	$\Pr[Z = 0]$	$\Pr[Z \leq 2]$
r6-bch-128	23,746	23,869	67	123	7627	$\approx 70.7\%$	$\approx 99.5\%$
r6-bch-192	45,194	45,197	101	3	7	$\approx 99.3\%$	$\approx 100.0\%$
r6-bch-256	69,252	69,259	133	7	29	$\approx 98.7\%$	$\approx 100.0\%$
r6-rmrs-128	20,480	20,553	67	73	2702	$\approx 78.8\%$	$\approx 99.8\%$
r6-rmrs-192	38,912	38,923	101	11	67	$\approx 97.2\%$	$\approx 100.0\%$
r6-rmrs-256	59,904	59,957	133	53	1432	$\approx 88.9\%$	$\approx 100.0\%$
r7-rmrs-128	17,664	17,669	66	5	16	$\approx 98.1\%$	$\approx 100.0\%$
r7-rmrs-192	35,840	35,851	100	11	67	$\approx 97.0\%$	$\approx 100.0\%$
r7-rmrs-256	57,600	57,637	131	37	704	$\approx 91.9\%$	$\approx 100.0\%$

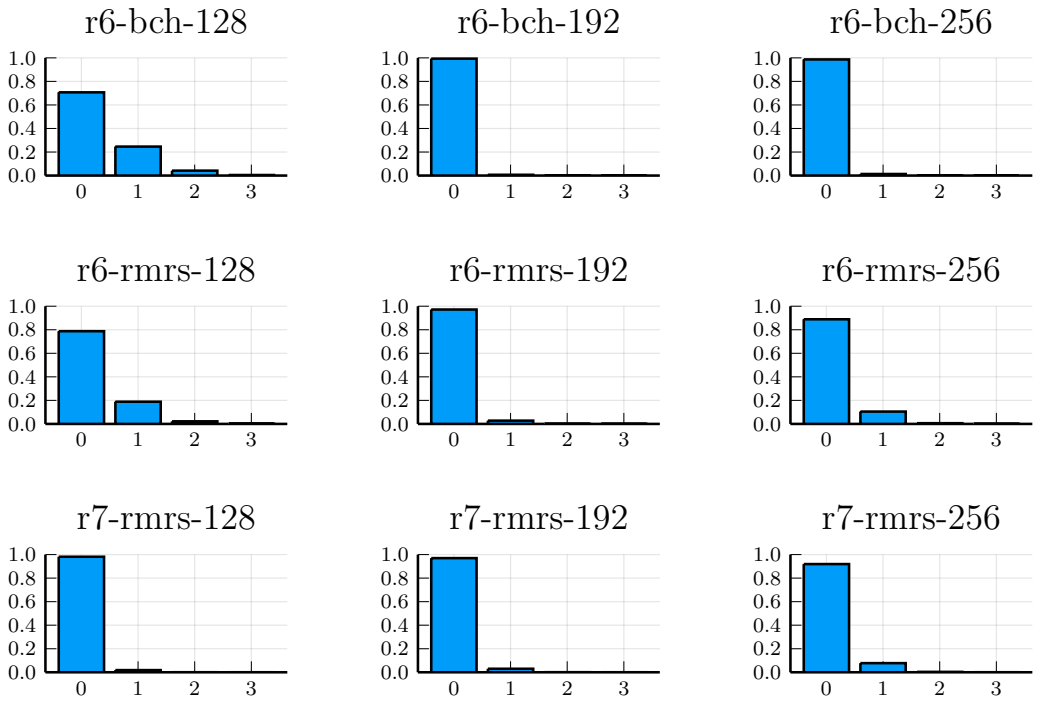


Figure 4.10: Probability distribution of $\Pr[Z = z]$ – the probability that $z \in \{0, 1, 2, 3\}$ bits do not land within the $n_1 n_2$ bits that can be recovered using attacks exploiting decoding failures.

CHAPTER 5

Empirical Evaluation

We empirically determine the success probability and number of idealized decapsulation oracle calls that the attacks we discovered require. Additionally we collect the number of bits that were misclassified. We obtain these data using the methods described in Section 5.2.1 and 5.2.2. To obtain idealized timing information for a given message we strip away everything except the sampling of random vectors. The `struct timing_info` in Fig. 5.1 contains perfect information on selected relevant aspects of the rejection sampling function. In particular, for a given message $\mathbf{m}$, it yields the number of `seedexpander` calls that the message results in.

5.1 Results

We performed 4096 attacks using the idealized decapsulation timing oracle in $\approx$ 249 CPU core hours for hqc-r8-rmrs-128. Of these attacks 3778 succeeded which results in an empirical success rate of $\approx 92.24\%$. This success rate is lower than the expected $\approx 96.51\%$. The probability that we obtain 3778 successes or less assuming a success probability of $\approx 96.51\%$ and modeling the attacks as a Bernoulli process is $\approx 2^{-124}$. Therefore, the difference is not likely to be caused by random chance. The deviation may be rooted in invalid independence assumptions during the estimation of the success probability. Among the attacks that failed less than 9 bits were classified incorrectly. For $\approx 67\%$ of failed attacks only a single bit was incorrect. Therefore an additional brute-force step could further increase the success probability to $\approx 97.44\%$. The attack required a median of 3,013,467 idealized decapsulation oracle calls.

For hqc-r6-bch-128 we could use the optimized attack variant. In ≈ 96 CPU core hours we ran 163 attacks using the idealized decapsulation timing oracle. The empirical success rate was $\approx 96.7\%$. Among the attacks that failed a maximum of 3 bits were misclassified. The attack also required orders of magnitude fewer decapsulation oracle calls than the original attack. The median number of idealized decapsulation oracle calls is 19,942

```
struct timing_info message_timing(uint64_t *m) {
  uint8_t theta[SHA512_BYTES] = {0};
  uint64_t r1[VEC_N_256_SIZE_64] = {0};
  uint64_t r2[VEC_N_256_SIZE_64] = {0};
  uint64_t e[VEC_N_256_SIZE_64] = {0};
  // Computing theta
  sha3_512(theta, (uint8_t*) m, VEC_K_SIZE_BYTES);

  // Create seed_expander from theta
  AES_XOF_struct seedexpander;
  seedexpander_init(&seedexpander, theta, theta + 32,
    SEEDEXPANDER_MAX_LENGTH);
  struct timing_info ti1 = vect_set_random_fixed_weight(
    &seedexpander, r1, PARAM_OMEGA_R);
  struct timing_info ti2 = vect_set_random_fixed_weight(
    &seedexpander, r2, PARAM_OMEGA_R);
  struct timing_info ti3 = vect_set_random_fixed_weight(
    &seedexpander, e, PARAM_OMEGA_E);

  struct timing_info ti = (struct timing_info) {
    .inner_iters =
      ti1.inner_iters + ti2.inner_iters + ti3.inner_iters,
    .outer_iters =
      ti1.outer_iters + ti2.outer_iters + ti3.outer_iters,
    .seed_expander_iters =
      ti1.seed_expander_iters +
      ti2.seed_expander_iters +
      ti3.seed_expander_iters,
  };
  return ti;
}
```

Figure 5.1: Idealized timing oracle

among the attacks that finished executing. Two failed attacks aborted early, because inconsistent results from the timing oracle caused the list of possible errors to be reduced to zero. This could either be caused by incorrect information from the side-channel or the block having a weight $> e_{\text{max}}^{\text{blocks}}$.

5.2 Implementation

We cover two ways to execute the attack concurrently and collect the metrics we presented: whether the attack succeeded, how many bits were incorrect, and the number of idealized decapsulation oracle calls performed. We consider either running the attack on a single node with multiple cores or many nodes with with multiple cores.

5.2.1 Map Reduce

When working on a single node with many cores implementing a parallel map reduce is easily achieved in Rust using the rayon data-parallel library [Ray]. It exposes the `ParallelIterator` interface (or trait in Rust terminology) that offers many frequently used parallel operations. Rayon implements this interface for most data structures in the Rust standard library. This enables data structure specific optimizations for work load distribution. Additionally, for arbitrary iterators whose source is not supported by rayon it offers a `par_bridge` to convert any `Iterator` into a `ParallelIterator`. The parallel bridge is expected to be less performant when used instead of a data structure specific iterator, as it has to use a Mutex to lock the source iterator to obtain a new item. However, in our case we only iterate over a few thousand items and each map operation takes > 100 seconds, so overhead from the iteration itself is not a limiting factor. We make use of `par_bridge` to transform a stream of empty tuples into a parallel iterator. For each empty tuple we execute the binary implementing the attack. The binary outputs information about the attack run such as the number of iterations required, the number of decapsulation oracle calls or whether the attack succeeded. The Rust program parses this output stream and aggregates these statistics. Thanks to rayon, we execute as many attacks in parallel as the node we are working on has physical threads. The resulting code turns out to be very short thanks to the powerful abstractions provided by rayon. The implementation is shown in Listing 10. The first line produces a parallel iterator of

```
let trials = 1000; // for example
let iter = std::iter::repeat(()).take(trials).par_bridge();
let successes: i64 = iter
    .map(|_| sample_success().unwrap() as i64)
    .sum();
```

Listing 10: Entire data pipeline using the `rayon` data-parallel Rust library.

`trials` empty tuples. These are then mapped, each resulting in an execution of the attack and returning whether the attack was successful or not. We then compute the sum to obtain the total number of attacks that succeeded out of the `trials` attacks. Here, we only show the code for generating statistics about the success probability.

`par_iter` Data Race Unfortunately, we discovered a concurrency bug that occurs frequently when executing `par_iter` on many cores: only a subset of the cores are busy when the number of items to process is less than $2 \cdot t^2$ where t is the number of threads that rayon is using. We have reported the issue, which was caused by a data race between a `done` flag and a check whether a worker still has items to process. A maintainer has since committed a change that fixes this issue [Par]. Luckily, we were able to use `rayon::iter::repeatn` instead of `par_bridge` as a workaround, which is in fact more suited for our particular use-case.

5.2.2 Distributed Map Reduce

Since our workload requires compute only we can use many small instances with 2 vCPUs and only 512 MiB. Compared to a single large instance this presents a 4× cost reduction. Thus, to obtain the success probability and other performance metrics of the attack in a reasonable time-frame we employ a slurm cluster with virtual machines rented from AWS. Using their parallel cluster[Aws] CLI tool, setting up slurm with automatic scaling becomes relatively easy. Unfortunately automatic scaling does take some time, as instance provisioning takes a few minutes.

Driver Program We use a Rust implementation [Timb] of the differential dataflow [McS13] computation model for performing a distributed map reduce. Differential dataflow is based on timely dataflow [Mur+13]. To reduce over a stream of values, we need a custom type that is serializable though "Abomonation". This is achieved with `abomonation_derive` and shown in Listing 11. We then implement a combiner that

```
#[derive(Clone, Debug,
         PartialEq, Eq, PartialOrd, Ord, Abomonation)]
struct Statistic {
    success: i64,
    trials: i64,
}
```

Listing 11: Definition of the `Statistic` struct.

takes two Statistics and reduces them to one. Additionally, we define an identity element, that satisfies the following relation:

```
s.combine(&Statistic::identity()) == s
```

Listing 12: Equation the `Statistic` implementation has to satisfy.

Each worker produces its share of empty tuples to process. The implementation is shown in Section A.3. We can then map and reduce over a collection of empty tuples as shown in Listing 13. This results in a stream that we can inspect to print the aggregate results. Gathering information for other performance characteristics is done by modifying the `Statistic` struct and modifying the program output parser to collect more information. We do not include these modifications for brevity.

Running the Program Setting up a distributed map reduce is a lot more complicated since we have to set the program up on multiple machines. Timely dataflow supports such operation through a hostfile, passed as an argument to the resulting program. The hostfile contains the IP address and port of each node, ordered by the node ID. Using slurm's `SLURM_NODEID` environment variable we can collect this information using a

```
manages
  .map(|_| (0, run_attack().unwrap()))
  .reduce(|_key, input, output| {
    let mut s = Statistic::identity();
    for (elem, count) in input {
      for _ in 0..*count {
        s = s.combine(elem);
      }
    }
    output.push((s, 1isize));
  })
```

Listing 13: Entire data pipeline for collecting statistics from multiple run.

simple shell script from our cluster. The host file can then be distributed to the nodes either through a shared drive mounted via NFS or through slurm's `sbcast` broadcasting tool.

Since the libc versions used by Rust on the development system and the remote systems are incompatible, we statically link the musl libc. This is achieved by building for the linux musl target which is achieved by setting the target flag when building with `cargo`. We can then invoke the binary on all compute nodes using `srun`. The most important code snippets are shown in Section A.3.

FaaS An even cheaper approach that we considered was to use Functions as a Service offerings such as Microsoft Azure Functions. However, our initial evaluation did not find them to be suitable as they did not scale-out at a desirable speed. This resulted in many requests failing, as the resources were overused. Unfortunately, we were unable to find configuration options for the automatic scale-out, that would allow us to resolve this issue. A solution may be to switch to a different service provider offering greater scaling elasticity or configurable scaling policies, however we did not further pursue this avenue.

CHAPTER 6

Exact Timing Distribution

We now turn our attention to modeling the timing distribution. This enables us to show that a message with 4 `seedexpander` calls does not exist with overwhelming probability. Such a message would be useful for exploitation as it would reduce the number of false positives that $\mathcal{D}^{\mathrm{TB}(\mathsf{sk},\cdot)}(\mathbf{c},\cdot)$ returns.

6.1 Number of Rejection Sampling Iterations

The number of `seedexpander` calls is influenced by the number of iterations the rejection sampling algorithm requires. Therefore, we want to obtain the probability that the rejection sampling algorithm takes at least a specified number of iterations. The rejection sampling algorithm's goal is to sample a random vector of weight w. In each iteration the algorithm samples an index into the vector from $\{1,\dots,n\}$ and sets the bit at that position to 1. It stops when w bits have been set – or alternatively, when w distinct positions have been sampled. We refer to the distinct indices sampled from $\{1,\dots,n\}$ as distinct elements. To model this algorithm we define the random variable $X_{n,i}$ as the number of distinct elements after sampling i elements from $\{1,\dots,n\}$. When we sample a new element, the probability that the element collides with a previously sampled element depends on the number of distinct elements sampled so far. We now want to determine the probability of sampling w distinct elements after having sampled i elements. There are two ways of obtaining w distinct elements:

1. We have already had w distinct elements in the previous iteration, and the new i^{th} sampled element **collides** with an element sampled before. Since there are w distinct elements sampled so far, the probability of a collision is $\frac{w}{n}$.
2. There are $w-1$ distinct elements in the previous iteration, and the i^{th} sampled element **does not collide** with an element sampled previously. The probability

that the element does not collide with the $w-1$ distinct elements sampled so far is 1 when $w-1 \leq 0$, or $1-\frac{w-1}{n}$ otherwise. This can also be written as $1-\max(0, \frac{w-1}{n})$.

Formally, the probability of sampling w distinct elements after having sampled i elements is:

$$\Pr[X_{n,i}=w] = \begin{cases} 0 & \text{if } i < w \\ 1 & \text{if } w = i = 0 \text{ or } w = i = 1 \\ \frac{w}{n}\Pr[X_{n,i-1}=w] + & \\ (1-\max(0,\frac{w-1}{n}))\Pr[X_{n,i-1}=w-1] & \text{otherwise} \end{cases} \tag{6.1}$$

We can also illustrate the process as a grid-like graph, where an edge to a left child node signifies that a distinct element was sampled, and an edge to a right child node that a collision occurred. This graph is shown in Fig. 6.1. Writing down the resulting

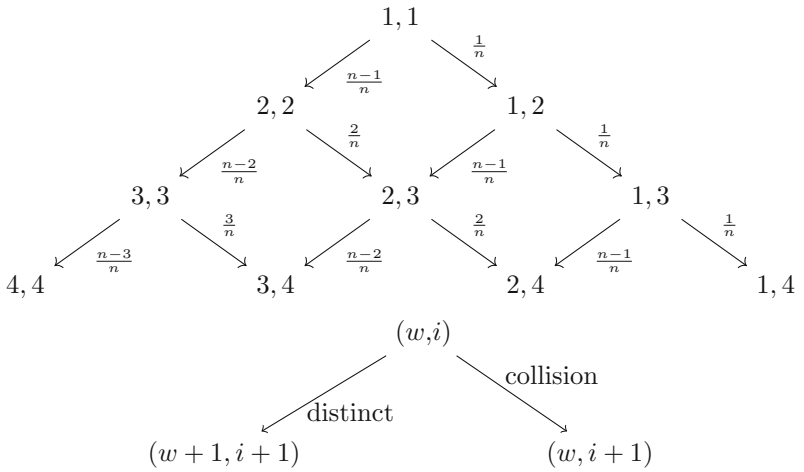


Figure 6.1: Graph of possible states in the rejection sampling algorithm and the transitions between them. Each node (w, i) represents that w distinct elements have been sampled after i iterations. The probability of reaching a node is the sum of the products of probabilities along every path to that node.

computations for individual nodes, we see a pattern emerge allowing us to express the above as an explicit formula:

$$\Pr[X_{n,i}=w] = \begin{cases} 0 & \text{if } i < w \\ \left(\prod_{x=1}^{w-1}\frac{n-x}{n}\right)\frac{\left\{{i \atop w}\right\}}{n^{i-w}} & \text{otherwise} \end{cases} \tag{6.2}$$

where $\left\{ {n \atop k} \right\}$ is the Stirling number of the second kind. However, the recursive formula fits our needs and may be computed efficiently using a Dynamic Programming (DP) algorithm. We additionally account for the possibility of failing to sample an element, which occurs with probability p_s. This models 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, 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 probability p_s. If a sample fails, it increases the iteration count, but no element is sampled. This yields:

$$\Pr[X_{n,i,p_s} = w] = \begin{cases} 0 & \text{if } i < w \\ 1 & \text{if } w = i = 0 \\ p_s & \text{if } w = i = 1 \\ p_s \dfrac{w}{n} \Pr[X_{n,i-1,p_s} = w] + & \\ (1 - p_s \max(0, \dfrac{w-1}{n})) \Pr[X_{n,i-1,p_s} = w-1] & \text{otherwise} \end{cases} \tag{6.3}$$

To compute $\Pr[X_{n,i,p_s} = w]$ we use the DP algorithm in Fig. 6.2. This is considerably faster than simply implementing the recursion in Eq. (6.3) in a naïve way. Alternatively, one can also cache the results during the recursion to obtain the same speedup. The DP algorithm essentially simulates a Markov chain, where a state is identified with a specific cell inside the ps matrix. It can also be seen as computing the probabilities of entire rows in the graph in Fig. 6.1. Given the probability of each node in a row, we can compute the probability of every node in the row below it.

Using these ways to compute the probability distribution of X_{n,i,p_s} 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:

$$\Pr[X_{n,i,p_s} \geq w] = \sum_{x=w}^{i} \Pr[X_{n,i,p_s} = x] \tag{6.4}$$

This cumulative probability can be efficiently computed from the table computed by the algorithm in Fig. 6.2.

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:

$$\Pr[U_{n,w,p_s} \leq i] = \Pr[X_{n,i,p_s} \geq w] \tag{6.5}$$

```
using OffsetArrays

function p_x(n, max_its, p_s)
  ps = OffsetArray(zeros(max_its+1, max_its+1), 0:max_its, 0:max_its)
  ps[0,0] = 1
  for i=0:max_its-1
    for w=0:i
      p_collision = min(w, n)/n
      success_and_not_collide = p_s * (1 - p_collision)
      ps[w+1, i+1] += ps[w, i] * success_and_not_collide
      ps[w  , i+1] += ps[w, i] * (1-success_and_not_collide)
    end
  end
  ps
end
```

Figure 6.2: Dynamic programming algorithm for computing a table ps whose w^{th} row and i^{th} column contains $\Pr[X_{n,i,p_s} = w]$.

6.2 Number of `seedexpander` Calls

The number of `seedexpander` calls performed depends upon the number of iterations the rejection sampling algorithm requires. In each iteration the rejection sampling algorithm consumes 3 bytes of the initially generated $3 \cdot \omega_{\mathbf{e}}$ bytes to generate $\mathbf{e}$, $\mathbf{r_1}$ or $\mathbf{r_2}$[1]. Given a target number of `seedexpander` calls ξ we can compute the number of rejection sampling iterations i that would result in this outcome. For $\xi = 0$, all iterations of sampling $\omega_{\mathbf{e}}$ distinct positions must be successful and $i = \omega_{\mathbf{e}}$. If $i > \omega_{\mathbf{e}}$, the randomness would be exhausted, and a call to `seedexpander` is made. The newly obtained randomness would suffice for another $\omega_{\mathbf{e}}$ iterations, or $2 \cdot \omega_{\mathbf{e}}$ in total. Thus, for $\xi = 1$ we know $\omega_{\mathbf{e}} + 1 \leq i \leq 2 \cdot \omega_{\mathbf{e}}$. In general for $\xi \geq 1$, we know that: $\xi \cdot \omega_{\mathbf{e}} + 1 \leq i \leq (\xi + 1) \cdot \omega_{\mathbf{e}}$. We define W to be the number of calls to the `seedexpander` function in the entirety of the decapsulation function. The probability that the decapsulation takes ξ `seedexpander` calls is:

$$\Pr[W_{n,\omega_{\mathbf{e}},p_s} = \xi] = \Pr[U_{n,\omega_{\mathbf{e}},p_s} \leq (\xi + 1) \cdot \omega_{\mathbf{e}}] - \Pr[U_{n,\omega_{\mathbf{e}},p_s} \leq \xi \cdot \omega_{\mathbf{e}}] \tag{6.6}$$

where n, $\omega_{\mathbf{e}}$ and p_s are the parameters for the specific HQC instance under scrutiny.

6.3 Total Number of `seedexpander` Calls

Extending the previous analysis we wish to know the probability distribution of the total number of `seedexpander` calls that are issued during a decapsulation. Each

[1] Note that the weight ω of $\mathbf{y}$ is different from $\omega_{\mathbf{e}} = \omega_{\mathbf{r}}$.

decapsulation performs 3 calls to vect_set_random_fixed_weight. Each of these calls can theoretically call seedexpander an unbounded number of times. This computation is analogous to computing the probability distribution of the sum of a set of dice: We have 3 dice, one for each call to vect_set_random_fixed_weight, with sides $\xi \in \mathbb{Z}_0$ corresponding to the number of seedexpander calls. We introduce the random variable $D_{d,\psi,P}$ which is the sum of d dice with sides $\xi \in \psi$ whose probability distribution is given by the random variable P. We define $\Pr[P = \xi] = 0$ for $\xi \notin \psi$. Its probability distribution is given by:

$$\Pr[D_{d,\psi,P} = \xi] = \begin{cases} \Pr[P = \xi] & \text{if } d = 1 \\ \sum_{\sigma \in \psi} \Pr[P = \sigma] \cdot \Pr[D_{d-1,\psi,P} = \xi - \sigma] & \text{otherwise} \end{cases} \tag{6.7}$$

Let the random variable T be the total number of seedexpander calls issued during the entire decapsulation. Since the sides of the dice are non-negative we only need to consider $\psi = \{0, \ldots, \xi\}$. Then, $\Pr[T = \xi] = \Pr[D_{3,\{0,\ldots,\xi\},W_{n,\omega_{\mathbf{e}},p_s}} = \xi]$. The probability distribution of T is displayed in Fig. 6.3 for numerous HQC parameters.

We experimentally verified our results for the hqc-r7-rmrs-128 parameter set with $10 \cdot 10^6$ samples, and find that they match our computations. The results may be viewed in Fig. 6.4. Using these results we can show that the probability of a message with ≥ 4 seedexpander calls is vanishingly small. The $\log_2$ probability of a message having ≥ 4 seedexpander calls is shown in Table 6.1. Assuming independence, the probability of a message existing which has ≥ 4 seedexpander calls is $1 - (1 - \Pr[T \geq 4])^{2^k}$ where k is the number of message bits.

Furthermore, we observe that the probability of 3 seedexpander calls is very close to the probability that every call to vect_set_random_fixed_weight results in a single seedexpander call. This is because the probability of a call to vect_set_random_fixed_weight resulting in ≥ 2 seedexpander calls is extremely low.

Table 6.1: The number of message bits k, the probability that a message has 3 `seedexpander` calls, the $\log_2$ of the probability that a message has 4 or more `seedexpander` calls, and the $\log_2$ of the probability of such a message existing.

Alias	k	$\Pr[T = 3]$	$\log_2(\Pr[T \geq 4])$	$\log_2\left(1 - (1 - \Pr[T \geq 4])^{2^k}\right)$
r6-bch-128	256	$\approx 0.8\%$	≈ -495	≈ -239
r6-bch-192	256	$\approx 0.7\%$	≈ -822	≈ -566
r6-bch-256	256	$\approx 2.0\%$	≈ -1052	≈ -796
r6-rmrs-128	256	$\approx 0.3\%$	≈ -530	≈ -274
r6-rmrs-192	256	$\approx 0.5\%$	≈ -840	≈ -584
r6-rmrs-256	256	$\approx 10.7\%$	≈ -904	≈ -648
r7-rmrs-128	128	$\approx 0.6\%$	≈ -486	≈ -358
r7-rmrs-192	192	$\approx 4.0\%$	≈ -695	≈ -503
r7-rmrs-256	256	$\approx 0.9\%$	≈ -1074	≈ -818

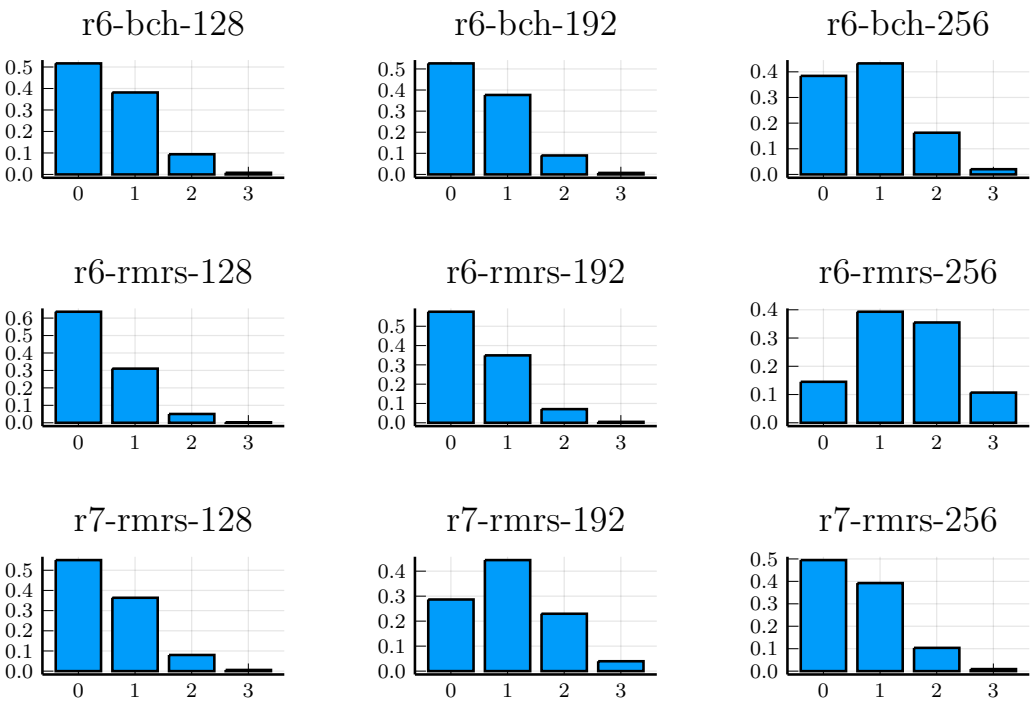


Figure 6.3: Computed probability distribution of the number of seedexpander calls in the for loop of the rejection sampling routine during decapsulation for random messages $\mathbf{m}$.

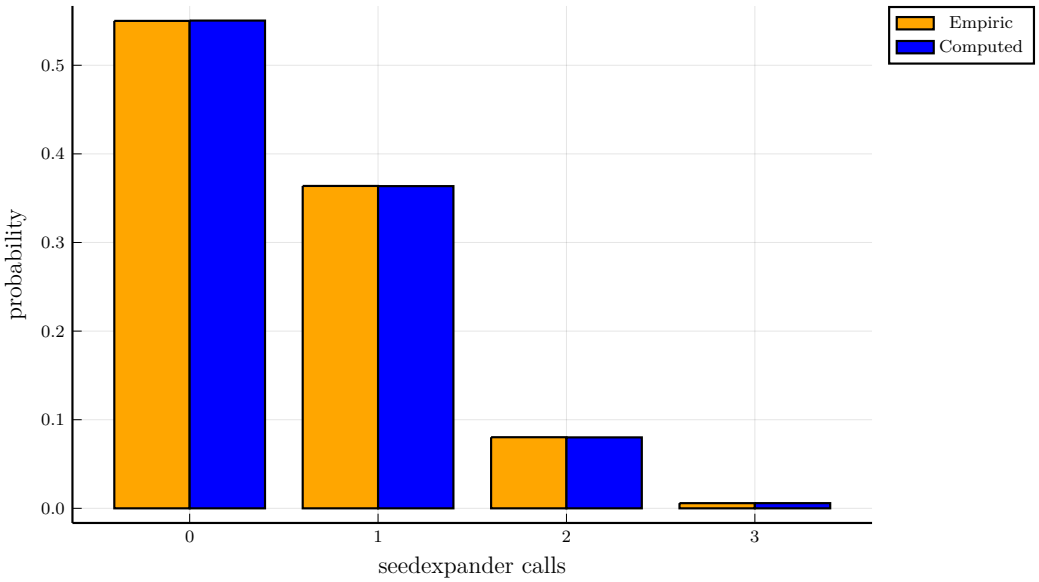


Figure 6.4: Probability distribution of the number of seedexpander in rejection sampling routine during decryption for random messages $\mathbf{m}$ for hqc-r7-rmrs-128. We verified the computed probabilities experimentally: they match within an error of less than 0.0004.

CHAPTER 7

Countermeasures

We now investigate countermeasures that could remove the side-channel.

7.1 No Additional `seedexpander` Calls

The first attempt we make at forging 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. As we have shown earlier in Section 6.3, it is vanishingly unlikely that the `seedexpander` function is called more than once within the for loop. Therefore, our first countermeasure is to increase the number of bytes that are generated initially to double the previous amount. This results in a 4 byte patch seen in Listing 14.

```
 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};
+     2*3 * PARAM_OMEGA_R] = {0};
```

Listing 14: No Additional `seedexpander` Calls Patch

We measure the timing distribution of HQC again. The results may be viewed in Fig. 7.1 We now no longer find multiple modes. Additionally, we find that no message in our 10 million samples generates any additional `seedexpander` invocations. However, the algorithm is not constant time: rejection sampling still performs a different number of iterations depending on the message. The generated timing difference is small but

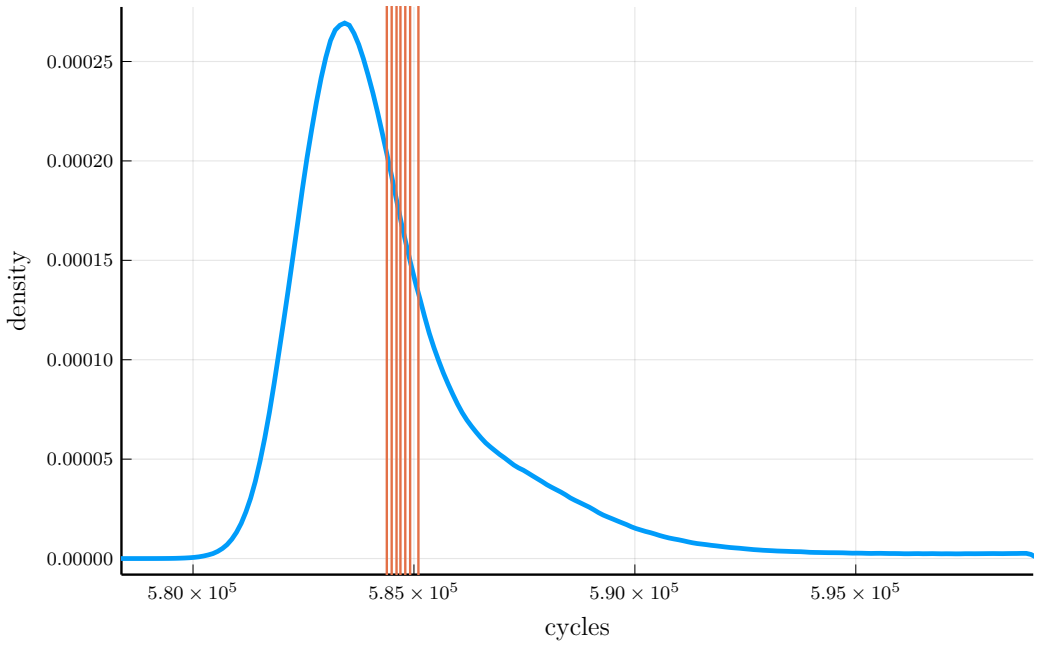


Figure 7.1: Probability density of the number of clock cycles required for the decapsulation function for random messages in hqc-r6-bch-128 with the `seedexpander` countermeasure applied. The vertical lines display the median of the conditional distributions illustrated in Fig. 7.2.

measurable, as can be observed in Fig. 7.2. It shows the probability density conditioned on the number of random indices generated during rejection sampling. The difference between the medians of the cycle count for each category varies between 88 and 187 cycles. While the countermeasure likely tremendously increases the effort required to perform the attack, it could still allow a local attacker to recover the key in $\approx 10^7 \frac{\text{samples}}{\text{idealized oracle call}} \cdot 10^4$ idealized oracle calls $= 10^{11} \approx 2^{37}$ samples. These are very coarse estimates, and we have not implemented the attack. However it still gives reason to look for alternatives that do not show any such leakage.

7.1.1 Constant-Time Random Number Generation

Before we look for completely different approaches, we look for further improvements. For example, we can remove the inner rejection sampling used for generating random indices into the vector. A constant-time RNG will also be useful for constant-time vector sampling algorithms. The inner rejection sampling can be seen in Algorithm 7.1.

Instead of rejection sampling integers in the range $0 \leq x < \left\lfloor \frac{2^k}{m} \right\rfloor n$ we can 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 integer if m does not divide 2^b, which is the case here.

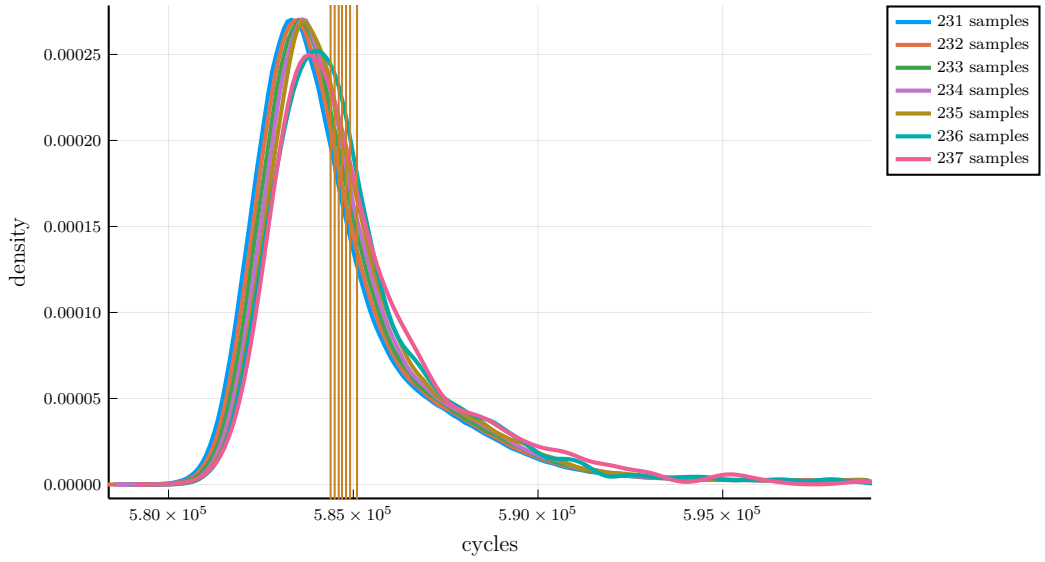


Figure 7.2: Probability density of the number of clock cycles required for the decapsulation function for random messages in hqc-r6-bch-128 with the seedexpander countermeasure applied; conditioned on the number of random indices generated.

Algorithm 7.1: Inner Rejection Sampling Algorithm

Result: Random number in $[0, \ldots, m-1]$

1 **repeat**
2 $\quad i \leftarrow\$ [0, 2^k)$
3 **until** $i < \left\lfloor \frac{2^k}{m} \right\rfloor m$
4 **return** $i \bmod m$

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

$$\mathrm{SD}_{X,Y} = \frac{1}{2} \cdot \sum_{z \in \Omega} |\Pr[X = z] - \Pr[Y = z]| \tag{7.1}$$

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

$$\Pr[U_m = z] = \begin{cases} \frac{1}{z} & \text{if } 0 \leq z < m \\ 0 & \text{otherwise} \end{cases} \tag{7.2}$$

Additionally, define the probability distribution M_n which reduces an integer in $\{0, \ldots, n-1\}$ modulo m. It's probability distribution is given by:

$$\Pr[M_n = z] = \begin{cases} \frac{\lfloor n/m \rfloor + 1}{n} & \text{if } 0 \leq z < n \bmod m \\ \frac{\lfloor n/m \rfloor}{n} & \text{if } n \bmod m \leq z < m \\ 0 & \text{otherwise} \end{cases} \tag{7.3}$$

The statistical distance between these two distributions is:

$$\begin{aligned} \mathrm{SD}_{U_m, M_n} &= \frac{1}{2} \cdot \sum_{z \in \{0,\ldots,m-1\}} |\Pr[U_m = z] - \Pr[M_b = z]| && (7.4) \\ &= \frac{1}{2} \cdot \left(\sum_{z \in \{0,\ldots,n \bmod m-1\}} |\Pr[U_m = z] - \Pr[M_b = z]| + \right. && (7.5) \\ &\qquad \left. \sum_{z \in \{n \bmod m,\ldots,m-1\}} |\Pr[U_m = z] - \Pr[M_b = z]| \right) \\ &= \frac{1}{2} \cdot \left(\sum_{z \in \{0,\ldots,n \bmod m-1\}} \left| \frac{1}{m} - \frac{\lfloor \frac{n}{m} \rfloor + 1}{n} \right| + \right. && (7.6) \\ &\qquad \left. \sum_{z \in \{n \bmod m,\ldots,m-1\}} \left| \frac{1}{m} - \frac{\lfloor \frac{n}{m} \rfloor}{n} \right| \right) \\ &= \frac{1}{2} \cdot \left((n \bmod m) \cdot \left| \frac{1}{m} - \frac{\lfloor \frac{n}{m} \rfloor + 1}{n} \right| + \right. && (7.7) \\ &\qquad \left. (m - (n \bmod m)) \cdot \left| \frac{1}{m} - \frac{\lfloor \frac{n}{m} \rfloor}{n} \right| \right) \end{aligned}$$

In Table 7.1 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. In the table we use $m = 23{,}869$ from hqc-r6-bch-128. Which statistical distance is acceptable is up to the designers of the scheme. For our further testing we use $b = 128$. We can implement a modular reduction of a 128 bit non-negative number x modulo a small number efficiently using basic rules of modular arithmetic. We can represent x in base 2^8:

$$x = x_0 + 2^8 \cdot x_1 + 2^{8 \cdot 2} \cdot x_2 + \cdots + 2^{8 \cdot (\ell - 1)} x_{\ell - 1} + 2^{8 \cdot \ell} \cdot x_\ell \tag{7.8}$$

We can then simplify the computation of $x \bmod m$ in the following way:

Table 7.1: 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-r8-rmrs-128.

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

$$x \bmod m = \left(\cdots \overbrace{\left(x_{\ell-1} + 2^8 \cdot \underbrace{(x_\ell \bmod m)}_{z_0} \right)}^{z_1} \bmod m \cdots \right) \bmod m \tag{7.9}$$

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

$$z_i = \begin{cases} x_\ell \bmod m & \text{if } i = 0 \\ (x_{\ell-i} + 2^8 \cdot z_{i-1}) \bmod m & \text{otherwise} \end{cases} \tag{7.10}$$

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 Listing 15.

```
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;
}
```

Listing 15: Reducing a number modulo a constant in multiple steps [pon]. `GEN_BYTES` is $\frac{b}{8}$.

Additionally, while a divide instruction is not constant-time in general on most 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 [MvV97, p.603]. This can be observed in Fig. 7.3. 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 AMD's Zen 2 ISA according to Agner's instruction tables [Agn]. To ensure that the compiled result always uses these

```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 7.3: Modular reduction of an integer a modulo a constant in C and the resulting intel-style assembly with optimization level 2 using clang.

instructions, which we have verified to be constant time, we can copy the compilation result into an `__asm__ volatile` block.

7.1.2 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 `vect_set_random_fixed_weight` function. We can use the random variable U_{n,w,p_s} defined in Section 6.1 to help us in this endeavour. The probability that a message emits ≥ 1 additional `seedexpander` calls when the randomness reservoir provides sufficient entropy for κ random indices is:

$$1 - (\Pr[U_{n,\omega_r,p_s} \leq \kappa])^3. \tag{7.11}$$

We would like this probability to be sufficiently low. We can compute 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 of success 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 7.2. 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 RNG still requires much more random bytes to be generated, since it requires 16 bytes per index, instead of ≈ 3 in expectation. The resulting code snippets are seen in Fig. 7.4 using the parameters obtained for hqc-r6-bch-128.

Table 7.2: Number of indices 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.

Alias	κ_{p_s}	$\log_2(1-(\Pr[U_{n,\omega_r,p_s} \leq \kappa])^3)$	κ_1	$\log_2(1-(\Pr[U_{n,\omega_r,1} \leq \kappa])^3)$
r6-bch-128	101	≈ -131	89	≈ -130
r6-bch-192	150	≈ -195	148	≈ -196
r6-bch-256	198	≈ -257	193	≈ -259
r6-rmrs-128	99	≈ -130	99	≈ -132
r6-rmrs-192	149	≈ -194	149	≈ -197
r6-rmrs-256	206	≈ -258	194	≈ -258
r7-rmrs-128	99	≈ -134	97	≈ -129
r7-rmrs-192	152	≈ -193	146	≈ -195
r7-rmrs-256	192	≈ -261	190	≈ -259

```
#define BYTES_PER_INDEX 3
#define K_PS 90
size_t random_bytes_size =
  BYTES_PER_INDEX * K_PS;
uint8_t rand_bytes[
    BYTES_PER_INDEX * K_PS
  ] = {0};
```

(a) Constant number of seedexpander calls

```
#define BYTES_PER_INDEX 16
#define K_1 87
size_t random_bytes_size =
  BYTES_PER_INDEX * K_1;
uint8_t rand_bytes[
    BYTES_PER_INDEX * K_1
  ] = {0};
```

(b) Constant number of seedexpander calls and constant-time random number generation

Figure 7.4: Code snippets for declaring the random byte reservoir and associated constant for hqc-r6-bch-128.

RNG optimization 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 $2^{8 \cdot 4}$.

```
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;
}
```

Further performance improvements may be achievable through the use of even wider registers or SIMD instructions.

7.1.3 Monte-Carlo Constant-Time

We can 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 7.2 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 using the variables shown in Listing 16.

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

Listing 16: Newly added variables to record whether a sampled bit position is needed in the resulting vector.

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 as shown in Listing 17.

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

Listing 17: For each sampled bit position we record whether the bit position is needed. `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 AVX instructions to set the sampled bit positions in the bit-array. This algorithm is vectorized to process the bit-array in 256 bit chunks and shown in Listing 18.

```
for (uint32_t i = 0 ; i < weight ; i++) {
  // we store the bloc number and bit position of each vb[i]
  uint64_t bloc = tmp[i] >> 6;
  bloc256[i] = _mm256_set1_epi64x(bloc >> 2);
  uint64_t pos = (bloc & 0x3UL);
  __m256i pos256 = _mm256_set1_epi64x(pos);
  __m256i mask256 = _mm256_cmpeq_epi64(pos256,posCmp256);
  uint64_t bit64 = 1ULL << (tmp[i] & 0x3f);
  __m256i bloc256 = _mm256_set1_epi64x(bit64);
  bit256[i] = bloc256&mask256;
}

for (uint32_t i = 0 ; i < LOOP_SIZE ; i++) {
  __m256i aux = _mm256_setzero_si256();
  __m256i i256 = _mm256_set1_epi64x(i);

  for (uint32_t j = 0 ; j < weight ; j++) {
    __m256i mask256 = _mm256_cmpeq_epi64(bloc256[j],i256);
    aux ^= bit256[j] & mask256;
  }

  _mm256_storeu_si256(&tmp256[i],
                      _mm256_xor_si256(tmp256[i],aux));
}
```

Listing 18: Setting of the sampled bit positions using AVX instructions.

We wish to modify this algorithm to only set bits if they are needed. The algorithm performs the following steps:

1. For each sampled *position*
   a) Set `bloc256[i]` to a vector containing the 256 bit block number of the position: $\left\lfloor \frac{position}{256} \right\rfloor$
   b) Set `bit256[i]` to a 256 bit vector containing a set bit at *position* mod 256
2. For each 256 bit chunk in the bit-array

a) Compute a 256 vector containing set bits at all positions that are inside the chunk:

b) Set `aux` = 0^{256}

c) For each $v \in$ `bit256`

   i. Set `mask256` = 1^{256} if position

   ii. Set `aux` = `aux` $\oplus$ (`mask256` $\wedge$ v)

d) Store the 256 vector in the bit-array

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 bitmask in Listing 19.

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

Listing 19: Addition of `take256` into the `bit256[i]` bitmask to ensure a bit is only added if it is needed.

This change results in the bitmask `bit256[i]` being 0^{256} if the bit is not needed. When this 256 bit vector is later xor'd with the `aux` variable, it will have no impact, since $0 \oplus x = x$. The full source listing of this final mitigation can be found in Section A.5.

7.1.4 Side-Channel Evaluation

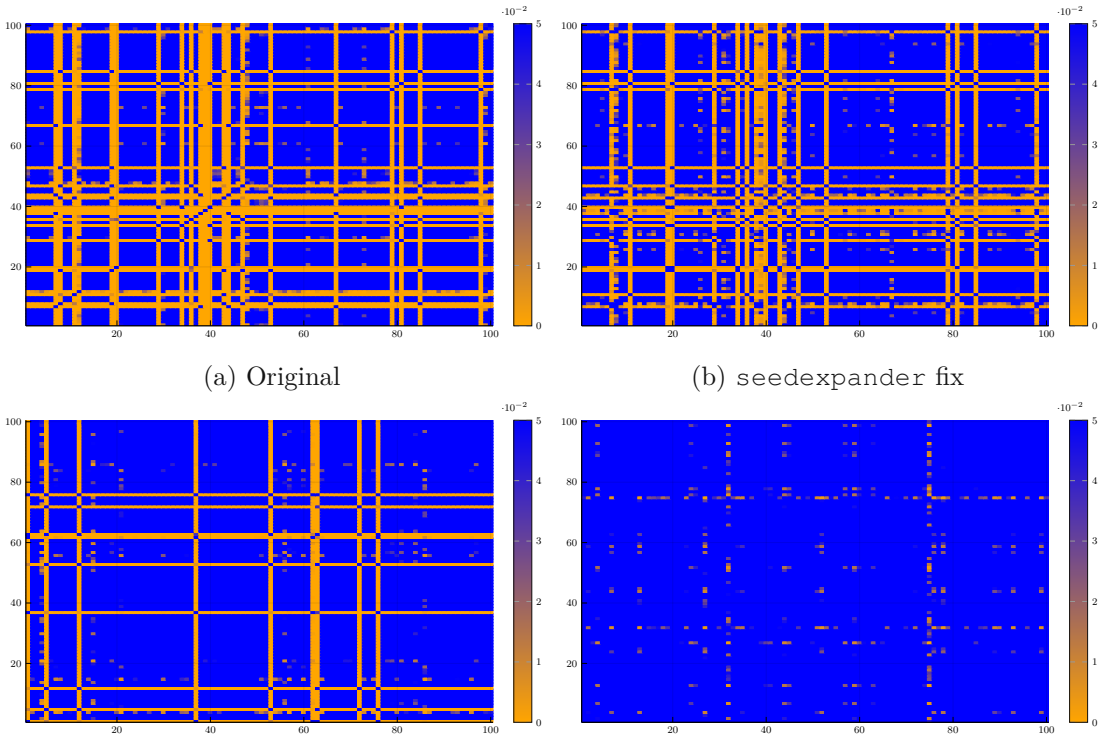
We perform the same test we performed in Section 4.1.3, however this time we only time the `vect_set_random_fixed_weight` function.

The results can be viewed in Fig. 7.5. The number of the detected difference clearly diminishes as more of the suggested modifications are applied. In particular, the final countermeasure eradicates all highly significant differences as can be seen in Fig. 7.5d. The remaining differences are likely noise: out of the 100^2 pairs tested, $\approx 4\%$ show a p-value of ≤ 0.05, which is in line with the expected 5%. The lowest p-value we observe is ≈ 0.017 with a median of ≈ 0.029 among those for which a statistically significant ($p \leq 0.05$) difference was detected. This is many orders of magnitude larger than for countermeasure in Fig. 7.5c. Its lowest p-value is $\approx 3 \cdot 10^{-26}$ with a median of $3 \cdot 10^{-9}$.

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.

7.1.5 Performance Evaluation

We measure the number of cycles the decapsulation function requires for random messages for the original and the two patched versions. We obtained 1 million measurements



(a) Original (b) seedexpander fix

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

Figure 7.5: P-values for Welch's t-test testing whether there is a statistically significant difference between the computation time of the seedexpander function for different seeds. Orange indicates that a statistically significant difference was detected.

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 server. The process is pinned to a single core, and all other processes are pinned to a different core. The results may be seen in Table 7.3. 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 induces a 36.4% 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 +1.5% 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

Version	Mean Cycles	Median Cycles
original	558,708	555,280 (+ 0.0%)
`seedexpander` fix	571,357	563,376 (+ 1.5%)
`seedexpander` + RNG fix	766,675	757,658 (+36.4%)
`seedexpander` + RNG fix + constant iters	791343	782,936 (+41.0%)

Table 7.3: Benchmark results in number of cycles for the modifications of the rejection sampling algorithm. Modifications tested on hqc-r6-bch-128. Cycle counts include the entirety of the decapsulation function.

not be a practical issue.

7.2 Constant Time

We now look for different algorithms, whose runtime is independent of the used randomness, generating a random vector of length n with a given weight w. We discuss two solutions, their potential and their drawbacks.

A simple approach is to fill the vector with w ones, and $n - w$ zeros and to then shuffle the array. This can be achieved by the Fisher-Yates shuffle [Knu97, p.145] whose modern implementation may be seen in Algorithm 7.2.

Algorithm 7.2: Fisher-Yates Shuffle

Input: Array v of n elements
Result: Array v is shuffled randomly
1 **for** $i \in \{0, \ldots, n-2\}$ **do**
2 $\quad j \leftarrow\$ \{0, \ldots, n-1\} + j$
3 $\quad$ Swap v_i and v_j
4 **end**

However, a naïve implementation of a Fisher-Yates shuffle may leak timing information, as it's using secret dependent array accesses. For example, if the memory location of v_j and $v_{j'}$ in two successive iterations fall into the same cache-line, the second access will have that memory location cached. However if v_j and $v_{j'}$ do not fall into the same cache-line the second memory access is much less likely to be cached and will thus take more time. This results in different timing profiles, depending upon the used seed, which would again be exploitable in HQC as it would again allow us to distinguish whether a ciphertext decrypts to a specific message or not. As a mitigation we can touch every element of the array whenever a single item is swapped, and place the element at the correct position using constant-time bit slicing operations. Such an algorithm may be viewed in Listing 20. The algorithm replaces the value at `v[j]` by `v[i]` and simultaneously stores the value at `v[j]` in the `val_j` variable. In each iteration it computes whether the current index `k` matches the desired index `j`.

1. Retrieving `v[j]`

   If the index matches, i.e. when `k = j`, `val_j` is overwritten to contain the value at the current index `v[k]`. Otherwise, the previous value of `val_j` is retained.

2. Placing `v[i]` at `v[j]`

   If the index matches, i.e. when `k = j`, we store the item `v[i]` at `v[k]`. Otherwise, we retain the value that was previously at `v[k]`.

Note that these operations must be performed in order, as the second operation will impact the outcome of the first. The result after the for loop is that `v[i]` has been placed at `v[i]` and the value that was previously at `v[j]` is now stored in `val_j`. We finish the swap algorithm by storing `val_j` at `v[i]`.

```
void const_time_swap(uint8_t *v, size_t n,
                     uint32_t i, uint32_t j) {
 uint8_t val_j = 0;
 for (uint32_t k = i; k < n; ++k) {
  uint8_t correct_idx = (k == j);
  val_j = (correct_idx & v[k]) | ((!correct_idx) & val_j);
  v[k]  = (correct_idx & v[i]) | ((!correct_idx) & v[k]);
 }
 v[i] = val_j;
}
```

Listing 20: Constant time swap of indices `i` and `j` in `v`, where `j` is secret but `i` is public.

We can use this algorithm in the Fisher-Yates implementation by calling `const_time_swap` with `j` set to the secret randomly sampled index. This results in a complexity of $\mathcal{O}(n^2)$ "operations" as we have to loop over the the remainder of the array for each element in the array. Since $n \geq 23{,}000$ is rather large an $\mathcal{O}(n^2)$ algorithm is rather prohibitive. We have tested a simple implementation and a single run of the `vect_set_random_fixed_weight` function alone takes on the order of $\approx 1.63 \cdot 10^9$ cycles. This is immensely slower than the original version and likely cannot be improved significantly without resorting to an algorithm with a lower asymptotic complexity. Additionally, the implementation we test is non-constant time, as we would need a constant-time modular reduction for every modulus in $\{2, \ldots, n\}$. This is feasible, but would incur a large additional code-size cost. The implementation of the non-constant time algorithm may be viewed in Section A.4.

Wang, Szefer, and Niederhagen propose to use a kind of reverse merge-sort algorithm implemented on an FPGA [WSN18]. As with the Fisher-Yates algorithm, we initialize an array with w ones and $n - w$ zeros: $1^w 0^{n-w}$. We then associate each bit v_i with a random number r_i: $((r_1, v_1), (r_2, v_2), \ldots, (r_n, v_n))$. Finally, we sort these pairs by the

random numbers using a merge-sort algorithm. We then discard the random numbers and keep the bits b_i in the new order. We now have a random permutation of the bits v_i. The algorithm has a complexity of $\mathcal{O}(n \log_2 n)$. The authors state that the distribution over the permutations is slightly biased, depending on the choice of b. To remove the bias they propose restarting the algorithm. However, this countermeasure would introduce an enormous timing-leak, and would be exploitable in HQC. Therefore, for our purposes, b must be chosen sufficiently large to reduce the bias in the permutation to a negligible factor. Furthermore, it is not clear how to implement the merge operation of the merge-sort algorithm in constant time. The merge operation performs memory accesses in an order dependent upon the values in the two arrays that are being merged. This could at least result in a timing side-channel when the adversary has the capability to probe the cache state.

Constant-time sorting networks could remedy the situation. Additionally Beneš permutation network has been suggested to use for such a task [BCS13]. Algorithms with an asymptotic complexity of $\mathcal{O}(n \log_2 n)$ or better could prove to be a fruitful avenue, as the asymptotic complexity of the algorithm implemented in HQC is $\mathcal{O}(n \cdot w)$ and typically $\log_2 n$ is smaller than w. However, the unknown constant factors could turn the tides either way.

In summary, the Fisher-Yates algorithm could be made constant-time but it would likely perform terribly. The merge-sort algorithm may perform as well or better than the solution implemented in HQC, however we are not aware of an efficient implementation of the merge operation that is constant-time.

CHAPTER 8

A Generic Timing Leak?

The timing leak we have identified could occur in multiple implementations of code-based cryptography. If the encryption algorithm uses rejection sampling and is using the Fujisaki-Okamoto transform [HHK17] the implementation may be vulnerable.

In fact, we have determined that a similar timing-variation exists in BIKE [Ara+], another code-based post-quantum KEM. Concretely, the decapsulation function in BIKE calls `function_h` using the decrypted message `m_prime`. In `function_h` the message `m_prime` is converted to and used as a seed. In `generate_error_vector` the seed is used to initialize a Pseudorandom Function (PRF) that is used as a RNG. `generate_error_vector` then calls out to `generate_indices_mod_z` to generate unique indices using the RNG and a rejection sampling algorithm. The `get_rand_mod_len` function uses the RNG to generate an index. The `aes_ctr_prf` function has a fast path for when the `prf_state` already has sufficiently many bytes in a reserve, and only performs AES invocations when the reserve has insufficient randomness. The relevant implementation details are listed in Section A.6.

We have tested the implementation and it does emit a different number of AES invocations for different ciphertexts. However, that does not directly imply that the implementation is vulnerable. The decoder in BIKE returns the error that needs to be corrected to obtain a codeword. If this error does not exactly have weight T it is replaced by a randomly sampled one. Later, the `function_h` is called with the `e_tmp` that is either the original error or the random error. Thus the side-channel would reveal whether the weight of the error is equal to T. The exploit, if at all possible, may look radically different for BIKE, as the scheme is not structurally similar to HQC.

If we can flip pairs of bits in the word the decoder obtains we could check if the error weight remains T. If it does, we know that exactly one of the flipped bits was an error bit. We can determine which bit is erroneous since we know that most bits are not error bits. Given that the pair of bits (a, b) contains one error, we check whether (a, c) and

(b, c) contains a single error for some random bit position c. Depending on the outcome of these we can determine whether a or b is the error bit, given that either a or b is erroneous. Note that the conclusion only follows under the assumption that c is not

flipping (a, c)	flipping (b, c)	Conclusion
weight $= T$	weight $= T$	impossible: $(a \oplus b) \wedge (a \oplus c) \wedge (b \oplus c) = \bot$
weight $= T$	weight $\neq T$	b is erroneous
weight $\neq T$	weight $= T$	a is erroneous
weight $\neq T$	weight $\neq T$	impossible: $(a \oplus b) \wedge \neg(a \oplus c) \wedge \neg(b \oplus c) = \bot$

Table 8.1: Conclusions that could be drawn from different oracle responses given an oracle that can tell whether the corrected error is of exactly weight T or not when flipping pairs of bits in the word that is to be decoded while assuming that bit c is error-free and either a or b is erroneous.

erroneous, however most bits are not erroneous. The possible outcomes are listed in Table 8.1. Unfortunately we cannot easily flip pairs of bits: the decoder is given $c_0 \cdot h_0$, where h_0 is secret and c_0 is part of the ciphertext. Flipping a bit in c_0 inserts entire copies of h_0 into the word that the decoder decodes.

We leave detailed investigation of the BIKE side-channel and potential attacks up for future work. We also note that BIKE is not designed for key-reuse, therefore timing-attacks may not be relevant to their attack model, unless they only require a single or very few decapsulation oracle queries. However, catastrophic failure of a cryptosystem when it is used in an environment it is not designed for is not a desirable property. Unfortunately, this regularly happens – either by accident or inexperience. For example GCM's unforgeability property completely falls apart when nonces are reused, yet nonce reuse and RNG failure has plagued many systems[1].

[1] A famous example is the Sony PS3 ECDSA signature flaw, where the same randomness was used for every signature, allowing secret key recovery from two signatures.

CHAPTER 9

Conclusion and Future Work

In this thesis we have identified a timing-side channel and forged two attacks exploiting it. We evaluated and improved their complexity and success probability. Our final success probability comes out at $\approx 92.24\%$ for the this attack and it functions on the all versions of HQC. It requires a median of 3,013,467 idealized decapsulation oracle calls.

Drawing from the state of the art in HQC timing attacks [WT+19] we constructed a new more efficient attack that drastically reduces the number of required decapsulation oracle calls. For this attack variant that functions on the BCH code versions of HQC we obtain a success probability of $\approx 96.7\%$. The optimized attack finished with a median of 19,942 idealized decapsulation oracle calls. However, the attack is tightly coupled to the codes used in HQC and thus has to be further adapted to newer versions of HQC using Reed-Muller codes instead of repetition codes.

Our research demonstrates that timing attacks are long from gone – in post-quantum cryptography they can shatter the security of a system even in the face of a classical attacker. Once such an issue has been identified the main goal for system designers and implementors is to determine a method to fix the vulnerability.

Luckily, the issues identified in this thesis are not a death-sentence for the cryptosystem, and can be remedied using the countermeasures we proposed. While our main countermeasure does have a heavy impact on total decapsulation time (+41%) we demonstrate that we cannot detect any statistically significant residual timing-variation in the patched version.

As auxiliary contributions we have identified an issue with the analysis set forth by Wafo-Tapa, Bettaieb, Bidoux, Gaborit, and Marcatel, which could cause a key recovery failure in up to $\approx 29\%$ of keys in hqc-r6-bch-128. To mitigate the issue we propose a low complexity brute-force search. Additionally, we have identified multiple minor issues in the paper that are to be fixed in later revisions.

Furthermore, we have discovered a data race in the rayon data-parallel Rust library and aided in the process of fixing the issue.

Future work could focus on the optimization of the countermeasure we propose through the use of SIMD instructions or different algorithms such as a merge sort or Benes permutation network. Additionally, the optimized attack could be adapted to the RMRS code version of HQC.

Lastly, we were able to identify a structurally similar issue in BIKE and leave detailed analysis and exploitation of it up for future work.

APPENDIX A

Source Listings

The following sections contain full source code listings for completeness sake.

A.1 Timing Data Collection

As an auxiliary data-structure to hold the timings and the result of the decapsulation, we introduce the timings struct that holds the CPU timestamps. The difference between two adjacent timestamps reveals the number of cycles spent in that part of the function.

```
#define SUB_TIMINGS 5
struct timings {
 uint64_t t[SUB_TIMINGS];
 int result;
};
```

The decapsulation function is divided into parts in the following way:

```
/**
 * @brief Decapsulation of the HQC_KEM IND_CAA2 scheme
 *
 * @param[out] ss String containing the shared secret
 * @param[in] ct String containing the ciphertext
 * @param[in] sk String containing the secret key
 * @returns 0 if decapsulation is successful, -1 otherwise
 */
```

```
struct timings crypto_kem_dec_timings(
 unsigned char *ss,
 const unsigned char *ct,
 const unsigned char *sk) {
 #ifdef VERBOSE
  printf("\n\n\n\n### DECAPS ###");
 #endif

 struct timings t;
 memset(&t, 0, sizeof t);

 int8_t result = -1;
 uint64_t u[VEC_N_256_SIZE_64] = {0};
 uint64_t v[VEC_N1N2_256_SIZE_64] = {0};
 unsigned char d[SHA512_BYTES] = {0};
 unsigned char pk[PUBLIC_KEY_BYTES] = {0};
 uint64_t m[VEC_K_SIZE_64] = {0};
 uint8_t theta[SHA512_BYTES] = {0};
 uint64_t u2[VEC_N_256_SIZE_64] = {0};
 uint64_t v2[VEC_N1N2_256_SIZE_64] = {0};
 unsigned char d2[SHA512_BYTES] = {0};
 unsigned char mc[
  VEC_K_SIZE_BYTES +
  VEC_N_SIZE_BYTES +
  VEC_N1N2_SIZE_BYTES
 ] = {0};

 // BEGIN PART 1 - Key loading
 t.t[0] = tic();
 // Retrieving u, v and d from ciphertext
 hqc_ciphertext_from_string(u, v , d, ct);

 // Retrieving pk from sk
 memcpy(pk, sk + SEED_BYTES, PUBLIC_KEY_BYTES);
 // END PART 1
 // BEGIN PART 2 - Decryption
 t.t[1] = toc();

 // Decryting [sic.]
 hqc_pke_decrypt(m, u, v, sk);
 // END PART 2
 // BEGIN PART 3 - Re-encryption
 t.t[2] = toc();
```

```
 // Computing theta
 sha3_512(theta, (uint8_t*) m, VEC_K_SIZE_BYTES);

 // Encrypting m'
  hqc_pke_encrypt(u2, v2, m, theta, pk);
 // END PART 3
 // BEGIN PART 4 - Shared secret derivation
 t.t[3] = toc();

 // Computing d'
 sha512(d2, (unsigned char *) m, VEC_K_SIZE_BYTES);

 // Computing shared secret
 memcpy(mc, m, VEC_K_SIZE_BYTES);
 memcpy(mc + VEC_K_SIZE_BYTES, u, VEC_N_SIZE_BYTES);
 memcpy(mc + VEC_K_SIZE_BYTES + VEC_N_SIZE_BYTES, v,
  VEC_N1N2_SIZE_BYTES);
 sha512(ss, mc,
  VEC_K_SIZE_BYTES + VEC_N_SIZE_BYTES + VEC_N1N2_SIZE_BYTES);

 // Abort if c != c' or d != d'
 result = (vect_compare(u, u2, VEC_N_SIZE_BYTES) == 0 &&
  vect_compare(v, v2, VEC_N1N2_SIZE_BYTES) == 0 &&
  vect_compare((uint64_t *)d, (uint64_t *)d2,
               SHA512_BYTES) == 0);
 for (size_t i = 0 ; i < SHARED_SECRET_BYTES ; i++) {
  ss[i] = result * ss[i];
 }
 result--;
 // END PART 4
 t.t[4] = toc();

 t.result = result;
 return t;
}
```

To obtain timings from the encryption function we return the timings struct from it and pass it through to the caller.

```
struct timings crypto_kem_dec_timings(/* [...] */) {
 // [...]
```

```
    // Encrypting m'
    struct timings t = hqc_pke_encrypt_timings(u2, v2, m,
                                                theta, pk);
    // [...]

    t.result = result;
    return t;
}
```

The encryption function is divided into parts in the following way:

```
/**
 * @brief Encryption of the HQC_PKE IND_CPA scheme
 *
 * The cihertext is composed of vectors <b>u</b> and <b>v</b>.
 *
 * @param[out] u Vector u (first part of the ciphertext)
 * @param[out] v Vector v (second part of the ciphertext)
 * @param[in] m Vector representing the message to encrypt
 * @param[in] theta Seed used to derive randomness required
 * for encryption
 * @param[in] pk String containing the public key
 */
struct timings hqc_pke_encrypt_timings(
    uint64_t *u, uint64_t *v,
    uint64_t *m,
    unsigned char *theta,
    const unsigned char *pk) {
    AES_XOF_struct seedexpander;
    uint64_t h[VEC_N_256_SIZE_64] = {0};
    uint64_t s[VEC_N_256_SIZE_64] = {0};
    uint64_t r1[VEC_N_256_SIZE_64] = {0};
    uint64_t r2[VEC_N_256_SIZE_64] = {0};
    uint64_t e[VEC_N_256_SIZE_64] = {0};
    uint64_t tmp1[VEC_N_256_SIZE_64] = {0};
    uint64_t tmp2[VEC_N_256_SIZE_64] = {0};
    struct timings t;
    memset(&t, 0, sizeof t);

    // BEGIN PART 1 - Init
    t.t[0] = tic();
    // Create seed_expander from theta
```

```
 seedexpander_init(&seedexpander, theta, theta + 32,
  SEEDEXPANDER_MAX_LENGTH);

 // Retrieve h and s from public key
 hqc_public_key_from_string(h, s, pk);
 // END PART 1
 // BEGIN PART 2 - Sample Vector
 t.t[1] = toc();

 // Generate r1, r2 and e
 vect_set_random_fixed_weight(&seedexpander, r1,
                              PARAM_OMEGA_R);
 vect_set_random_fixed_weight(&seedexpander, r2,
                              PARAM_OMEGA_R);
 vect_set_random_fixed_weight(&seedexpander, e,
                              PARAM_OMEGA_E);
 // END PART 2
 // BEGIN PART 3 - Encode
 t.t[2] = toc();

 // Compute u = r1 + r2.h
 vect_mul(u, r2, h);
 vect_add(u, r1, u, VEC_N_256_SIZE_64);

 // Compute v = m.G by encoding the message
 code_encode(v, m);
 vect_resize(tmp1, PARAM_N, v, PARAM_N1N2);
 // END PART 3
 // BEGIN PART 4 - Add error
 t.t[3] = toc();

 // Compute v = m.G + s.r2 + e
 vect_mul(tmp2, r2, s);
 vect_add(tmp2, e, tmp2, VEC_N_256_SIZE_64);
 vect_add(tmp2, tmp1, tmp2, VEC_N_256_SIZE_64);
 vect_resize(v, PARAM_N1N2, tmp2, PARAM_N);
 // END PART 4
 t.t[4] = toc();

 return t;
}
```

A.2 Optimized Attack

Obtaining a message with 3 seedexpander calls in up to 10^6 attempts. Empirically, such a message is found very quickly.

```
find_message(m, 3, 1000000);
struct timing_info ti1 = message_timing(m);
code_encode(v, m);
```

Main secret key recovery loop that is executed twice with different start and end positions for the blocks that are corrupted and recovered.

```
int corrupt_start = 0;            // inclusive
int corrupt_end = PARAM_DELTA;    // exclusive
int recover_start = PARAM_DELTA; // inclusive
int recover_end = PARAM_N1;       // exclusive
// Construct ciphertext with delta erroneous blocks
for (int i = corrupt_start; i < corrupt_end; ++i)
{
 for (int j = 0; j < PARAM_N2; ++j)
 {
  flip_bit(v_bytes, blocks_order[i] * PARAM_N2 + j);
 }
}
```

After corrupting δ blocks we recover the remaining blocks.

```
for (int i = recover_start; i < recover_end; ++i)
```

Generating all considered error patterns.

```
Vecs error_patterns = generate_error_patterns();
```

Struct definition of the `Vecs` type used for storing `len` instances of `Vec` that contain integers.

```
typedef struct {
 Vec *v;
 size_t len;
 size_t capacity;
} Vecs;
```

Struct definition of the `Vec` type used for storing `len` integers.

```
typedef struct {
 uint8_t *v;
 size_t len;
 size_t capacity;
} Vec;
```

Implementation of `generate_error_patterns` using the recursive helper function `vecs_of_weight_idx`. In each recursion depth all possible remaining positions are considered as candidates for bits to set. The maximum depth is `w_high`. The function generates all vectors whose weight is $\in \{$`w_low`$, \ldots,$ `w_high`$\}$.

```
Vecs generate_error_patterns() {
 Vecs vs = vecs_new();
 Vec v = vec_new();
 vecs_of_weight_idx(&vs, &v, PARAM_N2, 0, 0, MAX_E);
 return vs;
}

void vecs_of_weight_idx(
 Vecs *vecs, Vec *vec,
 int length, int start,
 int w_low, int w_high) {
 if (w_low <= 0 && 0 <= w_high) {
  vecs_push(vecs, vec_copy(vec));
  if (w_high == 0) {
   return;
  }
 }
 for (int i = start; i < length; ++i) {
  vec_push(vec, i);
  vecs_of_weight_idx(vecs, vec, length, i+1, w_low-1, w_high-1);
```

```
  vec_pop(vec);
 }
}
```

Generating random actions that will be evaluated for their efficiency in ruling out possible errors.

```
Vecs actions = sample_actions(3);
```

Rejection sampling of vectors of length n_2 with weight `weight` in the `sample_action` function. This function is called `n` times for each weight in $\{h - e^{\text{blocks}}_{\text{max}}, \ldots, h + e^{\text{blocks}}_{\text{max}}\}$.

```
Vec sample_action(size_t weight) {
 Vec v = vec_new();
 while (v.len < weight) {
  uint8_t pos = rand_mod(PARAM_N2);
  int exists = 0;
  for (size_t i = 0; i < v.len; ++i) {
   exists |= v.v[i] == pos;
  }
  if (!exists) {
   vec_push(&v, pos);
  }
 }
 return v;
}

Vecs sample_actions(int n) {
 Vecs vs = vecs_new();
 for (size_t w = H - MAX_E; w <= H + MAX_E; ++w) {
  for (int j = 0; j < n; ++j) {
   vecs_push(&vs, sample_action(w));
  }
 }
 return vs;
}
```

The combined weight function computes the weight of the vector $a \oplus b$, where the vectors a and b are given by their support, i.e. the positions of set bits.

```
size_t combined_weight(Vec a, Vec b) {
 // compute the weight if we were to add these two vectors
 size_t weight = a.len + b.len;
 // subtract 2 * intersection, since we counted them twice
 for (size_t i = 0; i < a.len; ++i) {
  for (size_t j = 0; j < b.len; ++j) {
   weight -= 2 * (a.v[i] == b.v[j]);
  }
 }
 return weight;
}
```

Applying the action to the **v** part of the ciphertext that is to be sent by flipping the respective bits the current block.

```
for (size_t j = 0; j < best_action->len; ++j)
{
 flip_bit(v_bytes,
  blocks_order[i] * PARAM_N2 + best_action->v[j]);
}
```

Calling the update error patterns function using the information deduced from the timing side-channel.

```
update_error_patterns(&error_patterns, *best_action, same_ti);
```

Function removing error patterns that are inconsistent with the result deduced from the timing information.

```
void update_error_patterns(
  Vecs *error_patterns, Vec action, int decoded_correctly)
{
 for (size_t error_idx = 0;
      error_idx < error_patterns->len; ++error_idx)
 {
  Vec error_pattern = error_patterns->v[error_idx];
  size_t weight = combined_weight(action, error_pattern);
```

```
  // Given the weight, we would expect
  int should_decode = weight < H;
  if (should_decode != decoded_correctly)
  {
   vecs_swap_remove(error_patterns, error_idx);
   --error_idx;
  }
 }
}
```

Counting the number of 1 votes and the total number of votes.

```
Vec pattern = error_patterns.v[0]; // solution
for (size_t j = 0; j < pattern.len; ++j)
{
  size_t pos = blocks_order[i] * PARAM_N2 + pattern.v[j];
  ++results[pos];
}
for (size_t j = 0; j < PARAM_N2; ++j)
{
  size_t pos = blocks_order[i] * PARAM_N2 + j;
  ++counters[pos];
}
```

Setting the block ranged anew for a second iteration of the entire block recovery procedure for each block in the remaining blocks.

```
corrupt_start = PARAM_DELTA;
corrupt_end = PARAM_DELTA * 2;
recover_start = 0;
recover_end = PARAM_DELTA;
```

Set the bits for which the 1 votes form a majority.

```
for (size_t i = 0; i < PARAM_N1N2; ++i)
{
 if (results[i] >= majority_min)
```

```
  {
   set_bit(recovered_y_bytes, i, 1);
  }
 }
```

Skipping blocks during the block recovery when a majority has already been formed for every bit.

```
int all_majority = 1;
for (size_t j = 0; j < PARAM_N2; ++j)
{
 size_t pos = blocks_order[i] * PARAM_N2 + j;
 if (results[pos] < majority_min &&
     (counters[pos] - results[pos]) < majority_min)
 {
  all_majority = 0;
  break;
 }
}
if (all_majority)
{
 continue; // Skip block
}
```

Action value function that computes the value an action has: the number of errors it can rule out in the worst case.

```
int64_t action_value(Vec action, Vecs error_patterns)
{
 // Compute the value of the action
 // Count how many error patterns we can rule out in the worst case
 int64_t decodes_count = 0;
 for (size_t error_idx = 0;
      error_idx < error_patterns.len; ++error_idx)
 {
  Vec error_pattern = error_patterns.v[error_idx];
  assert(error_pattern.len <= MAX_E);
 }
 for (size_t error_idx = 0;
      error_idx < error_patterns.len; ++error_idx)
```

```
  {
   Vec error_pattern = error_patterns.v[error_idx];
   size_t weight = combined_weight(action, error_pattern);

   int decodes = weight < H;
   decodes_count += decodes;
  }
  return min_int64(
   decodes_count,
   (error_patterns.len - decodes_count)
  );
}
```

A.3 Empirical Evaluation: Implementation

Implementation for the `Statistic` struct, which contains an identity and combination function:

```
impl Statistic {
  fn identity() -> Self {
    Self {
      success: 0,
      trials: 0,
    }
  }

  fn combine(&self, other: &Self) -> Self {
    Self {
      success: self.success + other.success,
      trials: self.trials + other.trials,
    }
  }
}
```

Inserting empty tuples as input:

```
let mut i = 0;
while i < num_samples {
  input.insert(());
```

```
  i += worker.peers();
}
```

Build command for compiling for the musl target.

```
cargo build --release --target x86\_64-unknown-linux-musl
```

Collecting hosts into a file, which the dataflow program requires.

```
srun bash -c 'echo $SLURM_NODEID $(hostname -i)' | \
  while read nodeid_nodeip; do
    echo ${nodeid_nodeip}:9999
  done | sort -n | awk '{print $2}' > hostfile.txt
```

Invoking the `gather_data` binary on all nodes using `srun`.

```
srun bash -c """
cd /tmp/driver
./gather_data $NUM_SAMPLES \
  -w 2              `# Number of workers per node` \
  -h hostfile.txt   `# IP + Port of each node` \
  -n $NUM_NODES     `# Total number of nodes` \
  -p \$SLURM_NODEID # This node's ID
"""
```

A.4 Fisher-Yates Countermeasure

```
void vect_set_random_fixed_weight_fisher_yates(
    AES_XOF_struct *ctx,
    uint64_t *v,
    uint16_t weight) {
#define BYTES_PER_INDEX 16
#define K_1 87
 uint8_t bits[PARAM_N];
 uint8_t rand_bytes[BYTES_PER_INDEX * PARAM_N];
 uint32_t j = 0;
```

```
 seedexpander(ctx, rand_bytes, sizeof rand_bytes);
 for (uint32_t i = 0; i < weight; ++i) {
  bits[i] = 1;
 }
 for (uint32_t i = weight; i < PARAM_N; ++i) {
  bits[i] = 0;
 }

 for (uint32_t i = 0; i <= PARAM_N-2; ++i) {
  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 - i; // not constant-time!
  }
  random_data += i;
  uint8_t val = bits[i];
  uint8_t other_val = 0;
  for (uint32_t j = i; j < PARAM_N; ++j) {
   uint8_t correct_idx = (j == random_data);
   other_val = (correct_idx & bits[j])
             | ((!correct_idx) & other_val);
   bits[j] = (correct_idx & val) | ((!correct_idx) & bits[j]);
  }
  // uint8_t tmp = bits[i];
  // bits[i] = bits[random_data];
  // bits[random_data] = tmp;
  bits[i] = other_val;
 }
 for (uint32_t i = 0; i < PARAM_N; ++i) {
  v[i >> 6] |= (-bits[i]) & (1 << (i & ((1 << 6)-1)));
 }
}
```

A.5 Constant-Time Monte-Carlo Sampling

```
void vect_set_random_fixed_weight(
    AES_XOF_struct *ctx,
    uint64_t *v,
    uint16_t weight) {
#define BYTES_PER_INDEX 16
#define K_1 87
```

```
size_t random_bytes_size = BYTES_PER_INDEX * K_1;
uint32_t rand_bytes[BYTES_PER_INDEX * K_1 / 4] = {0};
uint64_t random_data = 0;
uint32_t tmp[K_1] = {0};
uint8_t take[K_1] = {0};
uint32_t count = 0; // number of unique positions
uint8_t exist = 0;
size_t j = 0;
__m256i * tmp256 = (__m256i *) v;
__m256i bit256[K_1];
__m256i bloc256[K_1];
static __m256i posCmp256 = (__m256i){0UL,1UL,2UL,3UL};
#define LOOP_SIZE CEIL_DIVIDE(PARAM_N, 256)

seedexpander(ctx, (uint8_t*)rand_bytes, random_bytes_size);

for (uint32_t i = 0 ; i < K_1 ; ++i) {
 exist = 0;
 if (j == random_bytes_size) {
  seedexpander(ctx, (uint8_t*)rand_bytes, random_bytes_size);
  j = 0;
 }

 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;
 }

 for (uint32_t k = 0 ; k < i ; k++) {
  if (tmp[k] == random_data) {
   exist = 1;
  }
 }

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

```
 for (uint32_t i = 0 ; i < K_1 ; i++) {
  // we store the bloc number and bit position of each vb[i]
  uint64_t bloc = tmp[i] >> 6;
  bloc256[i] = _mm256_set1_epi64x(bloc >> 2);
  uint64_t pos = (bloc & 0x3UL);
  __m256i pos256 = _mm256_set1_epi64x(pos);
  __m256i mask256 = _mm256_cmpeq_epi64(pos256,posCmp256);
  uint64_t bit64 = 1ULL << (tmp[i] & 0x3f);
  __m256i bloc256 = _mm256_set1_epi64x(bit64);
  __m256i take256 = _mm256_set1_epi64x(take[i]) == 1;
  bit256[i] = bloc256&mask256&take256;
 }

 for (uint32_t i = 0 ; i < LOOP_SIZE ; i++) {
  __m256i aux = _mm256_setzero_si256();
  __m256i i256 = _mm256_set1_epi64x(i);

  for (uint32_t j = 0 ; j < K_1 ; j++) {
   __m256i mask256 = _mm256_cmpeq_epi64(bloc256[j],i256);
   aux ^= bit256[j] & mask256;
  }

  _mm256_storeu_si256(&tmp256[i],
                      _mm256_xor_si256(tmp256[i],aux));
 }

 #undef LOOP_SIZE
}
```

A.6 A Generic Timing Leak?

Call to `function_h` in the decapsulation function.

```
GUARD(function_h(&e_tmp, &m_prime));
```

Convert `m_prime` to a seed and use it to generate an error vector.

```
convert_m_to_seed_type(&seed, m);
return generate_error_vector(e, &seed);
```

The seed derived from m_prime is used to initialize a PRF that is used as a RNG.

```
GUARD(init_aes_ctr_prf_state(&prf_state,
                             MAX_AES_INVOKATION, seed));
```

Rejection sampling in BIKE for generating the support of the error vector using the RNG that is seeded by the seed derived from m_prime.

```
do {
 GUARD(get_rand_mod_len(&out[ctr], z, prf_state));
 ctr += is_new(out, ctr);
} while(ctr < num_indices);
```

Generating an index using the RNG.

```
GUARD(aes_ctr_prf((uint8_t *)rand_pos,
                  prf_state, sizeof(*rand_pos)));
```

Fast path in the RNG function for when there is sufficient randomness in the reserve.

```
// When Len is smaller than use what's left in the buffer,
// there is no need for additional AES invocations.
if((len + s->pos) <= AES256_BLOCK_BYTES) {
 bike_memcpy(a, &s->buffer.u.bytes[s->pos], len);
 s->pos += len;
 return SUCCESS;
}
```

List of Figures

List of Tables

List of Algorithms

Acronyms

AES
Advanced Encryption Standard 22, 93

AVX
Advanced Vector Extensions 87

BCH
Bose-Chaudhuri-Hocquenghem v, 16, 17, 27, 29, 30, 53, 95

BIKE
Bit Flipping Key Encapsulation 4, 5, 33, 93, 94, 96, 113

CCA
Chosen Ciphertext Attack 25, 29, *see* IND-CCA

CLI Command Line Interface 68

CPA
Chosen Plaintext Attack 29, *see* IND-CPA

CPU
Central Procesing Unit 5, 21, 22, 35, 45, 65, 68, 97

CRC
Cyclic Redundancy Check 8

DFR
Decryption/Decoding Failure Rate 4, 11, 26, 29, 30

DH Diffie-Hellman 2

DP Dynamic Programming 73

ECC
Error Correcting Code 1, 8, 25, 48, 53

ECDH
Elliptic-Curve Diffie-Hellman 2

FHT
Fast Hadamard Transform 15

FO Fujisaki Okamoto 4, 26

FPGA
Field-Programmable Gate Array 2, 91

GCM
Galois Counter Mode 94

HHK
Hofheinz Hövelmanns Kiltz 26, 29

HQC
Hamming Quasi-Cyclic v, 2–5, 11, 17, 18, 22–31, 33, 37, 39–43, 47, 48, 53, 59, 61, 62, 74, 75, 79, 82, 90, 92, 93, 95, 96, 115, 119

iid independent and identically distributed 38, 51

IND-CCA
Indistinguishability under Chosen Ciphertext Attack 19, 26, *see* IND-CCA

IND-CPA
Indistinguishability under Chosen Plaintext Attack 19, 26, 27, *see* IND-CPA

IP Internet Protocol 68

ISA Instruction Set Architecture 5, 23, 83

KEM
Key Encapsulation Mechanism 1–5, 18, 19, 25–29, 33, 42, 43, 93, 115, *see* KEM

LAC
Lattice-based Cryptosystems 29

MDPC
Moderate-Density Parity-Check 30

NFS
Network File System 69

NIST
National Institute of Standards and Technology 1, 2, 19, 45

OS Operating System 35

OW
One-Way *see* OW

OW-CPA
OW under CPA 19, 26, *see* OW-CPA

OWF
One-Way Function 18

PKE
Public Key Encryption 18, 19, 25–28, 38, 42, 115

PRF
Pseudorandom Function 93, 113, *see* PRF

QC Quasi-Cyclic 17, 18, 26, 30

QCSD
Quasi-Cyclic Syndrome Decoding 18, 26

RAM
Random Access Memory 8

RM Reed-Muller 3, 11, 12, 15, 27, 29, 30, 59–62, 95, 96

RNG
Random Number Generator 38, 80, 84, 85, 89, 90, 93, 94, 113

RS Reed-Solomon 3, 17, 27, 30, 53, 59, 60, 62, 96

SHA
Secure Hash Algorithm 44, 45

SIMD
Single Instruction Multiple Data 61, 86, 96

SMT
Satisfiability Modulo Theories 59

SSE
Streaming SIMD Extensions 22

T-OWF
Trapdoor One-Way Function 18, 19

TVLA
Test Vector Leakage Assessment 22

VM Virtual Machine 35

XOF
eXtendable-Output Function 38, 39, 44–46, 88, *see* XOF

Glossary

IND-CCA

Indistinguishability under Chosen Ciphertext Attack [KL14] is a security notion extending the IND-CPA notion. In addition to the encryption oracle the adversary is given access to a decryption oracle. Intuitively, the decryption oracle may be used by the adversary to decrypt ciphertexts related to the challenge ciphertexts returned by the encryption oracle. If the adversary manages to construct a valid ciphertext whose plaintext is related to the challenge ciphertext's plaintext, they may use the decryption oracle to leak information about the challenge plaintext.

The adversary again wins if they can determine b, however they may not query the decryption oracle on the challenge ciphertext.

IND-CPA

Indistinguishability under Chosen Plaintext Attack [KL14] is a security notion that an encryption scheme can fulfill. IND-CPA security is equivalent to semantic security which intuitively states that it is computationally infeasible to accurately compute any function of a plaintext given only the ciphertext.

Formally IND-CPA security is defined as a security game in which a probabilistic polynomial time adversary is given access to an encryption oracle. The adversary may send two messages m_0, m_1 to the challenger. The challenger returns the encryption $\mathrm{Enc}(m_b)$ where b is a uniform randomly chosen bit. The task of the adversary is to determine b given the ciphertext.

A scheme is IND-CPA secure if no probabilistic polynomial time adversary exists that could win this game with non-negligible advantage.

KEM

A Key Encapsulation Mechanism (KEM) [KL14, p.390f.] is a tuple of probabilistic polynomial-time algorithms $(\mathrm{KeyGen}, \mathrm{Encaps}, \mathrm{Decaps})$ such that:

$(pk, sk) \leftarrow \mathrm{KeyGen}(1^n)$

The key generation algorithm takes as input a string of length n, where n is the security parameter and outputs a public-/secret-key pair (pk, sk). As in [KL14] we assume that pk and sk both have length of at least n and that n can be determined from pk

$(c, k) \leftarrow \text{Encaps}(pk)$
: The encapsulation algorithm produces a ciphertext c and a key $k \in \{0,1\}^{\ell(n)}$ where $\ell(n)$ is the key-length.

$k \leftarrow \text{Decaps}(sk, c)$
: The decapsulation algorithm takes as input the ciphertext c and secret key sk and produces a key k.

A KEM is used as part of the KEM/DEM paradigm to encapsulate a symmetric key that is used by the Data Encryption Mechanism (DEM).

OW

A function f is one-way (OW) if no probabilistic polynomial time adversary adversary exists, who, given $f(x)$, can compute an x' with non-negligible probability such that $f(x) = f(x')$ [KL14].

In particular for all probabilistic polynomial time adversaries $\mathcal{A}$:

$$\Pr_x[\mathcal{A}(1^n, f(x)) \in f^{-1}(x)] = \mathsf{negl}(n) \tag{9.1}$$

where n is the security parameter.

OW-CPA

An encryption scheme is OW-CPA secure if there exists no probabilistic polynomial-time adversary who can determine a pre-image of $\text{Enc}(pk, m)$ with respect to $m \leftarrow\!\!\$\, \mathcal{M}$ with non-negligible probability, where $(pk, sk) \leftarrow\!\!\$\, \text{KeyGen}(1^n)$, $\mathcal{M}$ is the message space and n is the security parameter [HHK17].

PRF

A Pseudorandom Function (PRF) [KL14] $F_k(\cdot) : \{0,1\}^* \to \{0,1\}^*$ is a keyed function which is indistinguishable from a random function.

XOF

An eXtendable-Output Function is a function on bit-strings whose output may be extended to any desired length [KCP16]. When extending the length, the previous output is retained – i.e. when $f(x, 128) = a \| b$, then $f(x, 256) = a \| b \| c \| d$, where $\|$ denotes concatenation [CSD16].

Bibliography

[AFP13] Nadhem J. Al Fardan and Kenneth G. Paterson. "Lucky Thirteen: Breaking the TLS and DTLS Record Protocols". In: *2013 IEEE Symposium on Security and Privacy*. 2013 IEEE Symposium on Security and Privacy. 2013-05, pp. 526–540.

[Agn] Agner Fog. *Instruction Tables*. URL: https://www.agner.org/optimize/instruction_tables.pdf (visited on 2020-10-07).

[Agu+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. https://eprint.iacr.org/2016/1194. 2016.

[Agu+19] Carlos Aguilar Melchor, Nicolas Aragon, Slim Bettaieb, Loïc Bidoux, Olivier Blazy, Jean-Christophe Deneuville, Philippe Gaborit, Edoardo Persichetti, and Gilles Zémor. *HQC*. Tech. rep. available at https://csrc.nist.gov/projects/post-quantum-cryptography/round-2-submissions. National Institute of Standards and Technology, 2019.

[Aki+17] Koichiro Akiyama, Yasuhiro Goto, Shinya Okumura, Tsuyoshi Takagi, Koji Nuida, Goichiro Hanaoka, Hideo Shimizu, and Yasuhiko Ikematsu. *Giophantus*. Tech. rep. available at https://csrc.nist.gov/projects/post-quantum-cryptography/round-1-submissions. National Institute of Standards and Technology, 2017.

[Alb+20] Martin R. Albrecht, Daniel J. Bernstein, Tung Chou, Carlos Cid, Jan Gilcher, Tanja Lange, Varun Maram, Ingo von Maurich, Rafael Misoczki, Ruben Niederhagen, Kenneth G. Paterson, Edoardo Persichetti, Christiane Peters, Peter Schwabe, Nicolas Sendrier, Jakub Szefer, Cen Jung Tjhai, Martin Tomlinson, and Wen Wang. *Classic McEliece*. Tech. rep. available at https://csrc.nist.gov/projects/post-quantum-cryptography/round-3-submissions. National Institute of Standards and Technology, 2020.

[Alm+16] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, François Dupressoir, and Michael Emmi. "Verifying Constant-Time Implementations". In: 25th USENIX Security Symposium, 2016.

[AP16] Martin R. Albrecht and Kenneth G. Paterson. “Lucky Microseconds: A Timing Attack on Amazon’s S2n Implementation of TLS”. In: *Advances in Cryptology – EUROCRYPT 2016*. Ed. by Marc Fischlin and Jean-Sébastien Coron. Vol. 9665. Berlin, Heidelberg: Springer Berlin Heidelberg, 2016, pp. 622–643.

[Ara+] Nicolas Aragon, Paulo Barreto, Slim Bettaieb, Loïc Bidoux, Olivier Blazy, Jean-Christophe Deneuville, Philippe Gaborit, Shay Gueron, Tim Guneysu, Carlos Aguilar Melchor, Rafael Misoczki, Edoardo Persichetti, Nicolas Sendrier, Jean-Pierre Tillich, and Gilles Zémor. *BIKE: Bit Flipping Key Encapsulation*.

[Ara+19] 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, and Valentin Vasseur. *BIKE*. Tech. rep. available at `https://csrc.nist.gov/projects/post-quantum-cryptography/round-2-submissions`. National Institute of Standards and Technology, 2019.

[Ara+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*. Tech. rep. available at `https://csrc.nist.gov/projects/post-quantum-cryptography/round-3-submissions`. National Institute of Standards and Technology, 2020.

[ASK06] Onur Acıiçmez, Werner Schindler, and Çetin K. Koç. “Cache Based Remote Timing Attack on the AES”. In: *Topics in Cryptology – CT-RSA 2007*. Ed. by Masayuki Abe. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2006, pp. 271–286. DOI: 10.1007/11967668_18.

[ASY20] Emmanuel Abbe, Amir Shpilka, and Min Ye. *Reed-Muller Codes: Theory and Algorithms*. 2020-06-10. arXiv: 2002.03317 [cs, math]. URL: http://arxiv.org/abs/2002.03317 (visited on 2021-01-22).

[Aws] *Aws/Aws-Parallelcluster*. Amazon Web Services, 2021. URL: https://github.com/aws/aws-parallelcluster (visited on 2021-02-26).

[BB05] David Brumley and Dan Boneh. “Remote Timing Attacks Are Practical”. In: *Computer Networks* 48.5 (2005-08), pp. 701–716. ISSN: 13891286. DOI: 10.1016/j.comnet.2005.01.010. URL: https://linkinghub.elsevier.com/retrieve/pii/S1389128605000125 (visited on 2021-02-17).

[BCS13] Daniel J. Bernstein, Tung Chou, and Peter Schwabe. "McBits: Fast Constant-Time Code-Based Cryptography". In: *Cryptographic Hardware and Embedded Systems – CHES 2013*. Ed. by Guido Bertoni and Jean-Sébastien Coron. Vol. 8086. Lecture Notes in Computer Science. Santa Barbara, CA, USA: Springer, Heidelberg, Germany, 2013, pp. 250–272. DOI: 10.1007/978-3-642-40349-1_15.

[BE+06] Hagai Bar-El, Hamid Choukri, David Naccache, Michael Tunstall, and C. Whelan. "The Sorcerer's Apprentice Guide to Fault Attacks". In: *Proceedings of the IEEE* 94.2 (2006-02), pp. 370–382. ISSN: 1558-2256. DOI: 10.1109/JPROC.2005.862424.

[Bec+] G. Becker, J. Cooper, E. DeMulder, G. Goodwill, J. Jaffe, G. Kenworthy, T. Kouzminov, A. Leiserson, P. Rohatgi, and S. Saab. *Test Vector Leakage Assessment (TVLA) Methodology in Practice*.

[Ber05] Daniel J Bernstein. *Cache-Timing Attacks on AES*. 2005. URL: https://cr.yp.to/antiforgery/cachetiming-20050414.pdf (visited on 2021-02-17).

[Ber06] Daniel J. Bernstein. "Curve25519: New Diffie-Hellman Speed Records". In: *PKC 2006: 9th International Conference on Theory and Practice of Public Key Cryptography*. Ed. by Moti Yung, Yevgeniy Dodis, Aggelos Kiayias, and Tal Malkin. Vol. 3958. Lecture Notes in Computer Science. New York, NY, USA: Springer, Heidelberg, Germany, 2006, pp. 207–228. DOI: 10.1007/11745853_14.

[Ber+19] Daniel J. Bernstein, Tung Chou, Tanja Lange, Ingo von Maurich, Rafael Misoczki, Ruben Niederhagen, Edoardo Persichetti, Christiane Peters, Peter Schwabe, Nicolas Sendrier, Jakub Szefer, and Wen Wang. *Classic McEliece*. Tech. rep. available at https://csrc.nist.gov/projects/post-quantum-cryptography/round-2-submissions. National Institute of Standards and Technology, 2019.

[BH09] Billy Bob Brumley and Risto M. Hakala. "Cache-Timing Template Attacks". In: *Advances in Cryptology – ASIACRYPT 2009*. Ed. by Mitsuru Matsui. Vol. 5912. Lecture Notes in Computer Science. Tokyo, Japan: Springer, Heidelberg, Germany, 2009, pp. 667–684. DOI: 10.1007/978-3-642-10366-7_39.

[BMT78] Elwyn R. Berlekamp, Robert J. McEliece, and Henk C. A. van Tilborg. "On the Inherent Intractability of Certain Coding Problems (Corresp.)" In: *IEEE Transactions on Information Theory* 24.3 (1978-05), pp. 384–386. ISSN: 0018-9448. DOI: 10.1109/TIT.1978.1055873. URL: http://ieeexplore.ieee.org/document/1055873/.

[Bra+20] Lukas Brandstetter, Marc Fischlin, Robin Leander Schröder, and Michael Yonli. “On the Memory Fault Resilience of TLS 1.3”. In: *Security Standardisation Research*. Ed. by Thyla van der Merwe, Chris Mitchell, and Maryam Mehrnezhad. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2020, pp. 1–22. DOI: 10.1007/978-3-030-64357-7_1.

[Bru+16] 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: *Cryptographic Hardware and Embedded Systems – CHES 2016*. Ed. by Benedikt Gierlichs and Axel Y. Poschmann. Vol. 9813. Lecture Notes in Computer Science. Santa Barbara, CA, USA: Springer, Heidelberg, Germany, 2016, pp. 323–345. DOI: 10.1007/978-3-662-53140-2_16.

[Car+20] Carlos Aguilar Melchor, Nicolas Aragon, Slim Bettaieb, Loïc Bidoux, Olivier Blazy, Jurjen Bos, Jean-Christophe Deneuville, Jérôme Lacan, Edoardo Persichetti, Jean-Marc Robert, Pascal Véron, Philippe Gaborit, Arnaud Dion, and Gilles Zémor. “Hamming Quasi-Cyclic (HQC) Third Round Version Updated Version 10/01/2020”. In: (2020-10-01), p. 47. URL: https://pqc-hqc.org/doc/hqc-reference-implementation_2020-10-01.zip (visited on 2020-11-10).

[CFP17] Olive Chakraborty, Jean-Charles Faugère, and Ludovic Perret. *CFPKM*. Tech. rep. available at https://csrc.nist.gov/projects/post-quantum-cryptography/round-1-submissions. National Institute of Standards and Technology, 2017.

[CGL06] Denis Charles, Eyal Goren, and Kristin Lauter. *Cryptographic hash functions from expander graphs*. Cryptology ePrint Archive, Report 2006/021. https://eprint.iacr.org/2006/021. 2006.

[CK18] Natalie Collina and Zachary Kincaid. *Fastermind: Using a SAT-Solver to Play Mastermind More Efficiently*. 2018. URL: https://nataliecollina.com/wp-content/uploads/2019/12/IWSpringPaper.pdf (visited on 2021-05-08).

[CMM19] Tung Chou, Yohei Maezawa, and Atsuko Miyaji. “A Closer Look at the Guo-Johansson-Stankovski Attack Against QC-MDPC Codes”. In: *ICISC 18: 21st International Conference on Information Security and Cryptology*. Ed. by Kwangsu Lee. Vol. 11396. Lecture Notes in Computer Science. Seoul, Korea: Springer, Heidelberg, Germany, 2019, pp. 341–353. DOI: 10.1007/978-3-030-12146-4_21.

[CSD16] Information Technology Laboratory Computer Security Division. *SHA-3 2014 Workshop | CSRC*. CSRC | NIST. 2016-12-22. URL: https://csrc.nist.gov/events/2014/sha-3-2014-workshop (visited on 2021-05-14).

[CSD17a] Information Technology Laboratory Computer Security Division. *Post-Quantum Cryptography Standardization - Post-Quantum Cryptography | CSRC | CSRC*. CSRC | NIST. 2017-01-03. URL: https://csrc.nist.gov/projects/post-quantum-cryptography (visited on 2021-05-06).

[CSD17b] Information Technology Laboratory Computer Security Division. *Round 3 Submissions - Post-Quantum Cryptography | CSRC | CSRC*. CSRC | NIST. 2017-01-03. URL: https://csrc.nist.gov/projects/post-quantum-cryptography/round-3-submissions (visited on 2021-05-09).

[CSD17c] Information Technology Laboratory Computer Security Division. *Security (Evaluation Criteria) - Post-Quantum Cryptography*. CSRC | NIST. 2017-01-03. URL: https://content.csrc.e1a.nist.gov/Projects/Post-Quantum-Cryptography/Post-Quantum-Cryptography-Standardization (visited on 2021-05-06).

[D'A+19] Jan-Pieter D'Anvers, Angshuman Karmakar, Sujoy Sinha Roy, and Frederik Vercauteren. *SABER*. Tech. rep. available at https://csrc.nist.gov/projects/post-quantum-cryptography/round-2-submissions. National Institute of Standards and Technology, 2019.

[D'A+20] Jan-Pieter D'Anvers, Angshuman Karmakar, Sujoy Sinha Roy, Frederik Vercauteren, Jose Maria Bermudo Mera, Michiel Van Beirendonck, and Andrea Basso. *SABER*. Tech. rep. available at https://csrc.nist.gov/projects/post-quantum-cryptography/round-3-submissions. National Institute of Standards and Technology, 2020.

[Dan14] Daniel J. Bernstein. *[Cfrg] 25519 Naming*. 2014. URL: https://mailarchive.ietf.org/arch/msg/cfrg/-9LEdnzVrE5RORux3Oo_oDDRksU/ (visited on 2021-05-09).

[Din+20] Jintai Ding, Ming-Shing Chen, Albrecht Petzoldt, Dieter Schmidt, Bo-Yin Yang, Matthias Kannwischer, and Jacques Patarin. *Rainbow*. Tech. rep. available at https://csrc.nist.gov/projects/post-quantum-cryptography/round-3-submissions. National Institute of Standards and Technology, 2020.

[DMR11] Hang Dinh, Cristopher Moore, and Alexander Russell. "McEliece and Niederreiter Cryptosystems That Resist Quantum Fourier Sampling Attacks". In: *Advances in Cryptology – CRYPTO 2011*. Ed. by Phillip Rogaway. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2011, pp. 761–779. DOI: 10.1007/978-3-642-22792-9_43.

[Don77] Donald Ervin Knuth. "The Computer As Master Mind". In: *Journal of Recreational Mathematics* 9.1 (1977).

[Dra+18] Vlad Dragoi, Tania Richmond, Dominic Bucerzan, and Axel Legay. "Survey on Cryptanalysis of Code-Based Cryptography: From Theoretical to Physical Attacks". In: 2018 7th International Conference on Computers Communications and Control (ICCCC). Oradea: IEEE, 2018-05, pp. 215–223.

[DRV19] Jan-Pieter D'Anvers, Mélissa Rossi, and Fernando Virdia. *(One) failure is not an option: Bootstrapping the search for failures in lattice-based encryption schemes*. Cryptology ePrint Archive, Report 2019/1399. https://eprint.iacr.org/2019/1399. 2019.

[Dwo15] Morris J. Dworkin. "SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions". In: (2015-08-04). URL: https://www.nist.gov/publications/sha-3-standard-permutation-based-hash-and-extendable-output-functions (visited on 2020-11-07).

[Eat+18] Edward Eaton, Matthieu Lequesne, Alex Parent, and Nicolas Sendrier. "QC-MDPC: A Timing Attack and a CCA2 KEM". In: *Post-Quantum Cryptography - 9th International Conference, PQCrypto 2018*. Ed. by Tanja Lange and Rainer Steinwandt. Fort Lauderdale, Florida, United States: Springer, Heidelberg, Germany, 2018, pp. 47–76. DOI: 10.1007/978-3-319-79063-3_3.

[Eis+18] Kirsten Eisentraeger, Sean Hallgren, Kristin Lauter, Travis Morrison, and Christophe Petit. *Supersingular isogeny graphs and endomorphism rings: reductions and solutions*. Cryptology ePrint Archive, Report 2018/371. https://eprint.iacr.org/2018/371. 2018.

[Fab+17] Tomás Fabsic, Viliam Hromada, Paul Stankovski, Pavol Zajac, Qian Guo, and Thomas Johansson. "A Reaction Attack on the QC-LDPC McEliece Cryptosystem". In: *Post-Quantum Cryptography - 8th International Workshop, PQCrypto 2017*. Ed. by Tanja Lange and Tsuyoshi Takagi. Utrecht, The Netherlands: Springer, Heidelberg, Germany, 2017, pp. 51–68. DOI: 10.1007/978-3-319-59879-6_4.

[Gab10] Gabriele Paoloni. *How to Benchmark Code Execution Times on Intel® IA-32 and IA-64 Instruction Set Architectures*. 2010-09. URL: https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/ia-32-ia-64-benchmark-code-execution-paper.pdf (visited on 2021-02-06).

[GB+16] 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: Cryptographic Hardware and Embedded Systems – CHES 2016. Ed. by Benedikt Gierlichs and Axel Y. Poschmann. Vol. 9813. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2016, pp. 323–345. DOI: 10.1007/978-3-662-53140-2_16.

URL: http://link.springer.com/10.1007/978-3-662-53140-2_16 (visited on 2020-11-27).

[Gen+16] Daniel Genkin, Lev Pachmanov, Itamar Pipman, and Eran Tromer. "ECDH Key-Extraction via Low-Bandwidth Electromagnetic Attacks on PCs". In: *Topics in Cryptology - CT-RSA 2016*. Ed. by Kazue Sako. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2016, pp. 219–235. DOI: 10.1007/978-3-319-29485-8_13.

[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: *Advances in Cryptology – CRYPTO 2020, Part II*. Ed. by Daniele Micciancio and Thomas Ristenpart. Vol. 12171. Lecture Notes in Computer Science. Santa Barbara, CA, USA: Springer, Heidelberg, Germany, 2020, pp. 359–386. DOI: 10.1007/978-3-030-56880-1_13.

[GJS16] Qian Guo, Thomas Johansson, and Paul Stankovski. "A Key Recovery Attack on MDPC with CCA Security Using Decoding Errors". In: *Advances in Cryptology – ASIACRYPT 2016, Part I*. Ed. by Jung Hee Cheon and Tsuyoshi Takagi. Vol. 10031. Lecture Notes in Computer Science. Hanoi, Vietnam: Springer, Heidelberg, Germany, 2016, pp. 789–815. DOI: 10.1007/978-3-662-53887-6_29.

[Han] *Handbook of Information Security: Threats, Vulnerabilities, Prevention, Detection, and Management, Volume 3 [Book]*. 2006. URL: https://www.oreilly.com/library/view/handbook-of-information/9780471648321/ (visited on 2021-05-10).

[Har86] Harald Niederreiter. "Knapsack-Type Cryptosystems and Algebraic Coding Theory". In: *Problems of Control and Information Theory* 15.2 (1986), pp. 159–166. URL: http://real-j.mtak.hu/7997/1/MTA_ProblemsOfControl_15.pdf (visited on 2021-03-10).

[HGS99] Chris Hall, Ian Goldberg, and Bruce Schneier. "Reaction Attacks against several Public-Key Cryptosystems". In: *ICICS 99: 2nd International Conference on Information and Communication Security*. Ed. by Vijay Varadharajan and Yi Mu. Vol. 1726. Lecture Notes in Computer Science. Sydney, Australia: Springer, Heidelberg, Germany, 1999, pp. 2–12.

[HHK17] Dennis Hofheinz, Kathrin Hövelmanns, and Eike Kiltz. "A Modular Analysis of the Fujisaki-Okamoto Transformation". In: *TCC 2017: 15th Theory of Cryptography Conference, Part I*. Ed. by Yael Kalai and Leonid Reyzin. Vol. 10677. Lecture Notes in Computer Science. Baltimore, MD, USA: Springer, Heidelberg, Germany, 2017, pp. 341–371. DOI: 10.1007/978-3-319-70500-2_12.

[Hor07] Kathy Jennifer Horadam. *Hadamard Matrices and Their Applications*. Princeton University Press, 2007. ISBN: 978-0-691-11921-2. JSTOR: j.ctt7t6pw.

[HP03] William Cary Huffman and Vera Pless. *Fundamentals of Error-Correcting Codes*. 1st ed. Cambridge University Press, 2003-06-26. ISBN: 978-0-521-78280-7 978-0-521-13170-4 978-0-511-80707-7.

[Hqca] *HQC Submission Package 2020-05-29*. URL: https://pqc-hqc.org/doc/hqc-submission_2020-05-29.zip (visited on 2020-11-08).

[Hqcb] *HQC Submission Package 2020-06-06*. URL: https://pqc-hqc.org/download.php?file=hqc-submission_2021-06-06.zip (visited on 2020-11-08).

[Hqcc] *HQC Submission Package 2020-10-01*. URL: https://pqc-hqc.org/doc/hqc-submission_2020-10-01.zip (visited on 2020-11-08).

[Hqcd] *HQC Updates*. 2020. URL: https://groups.google.com/a/list.nist.gov/g/pqc-forum/c/htJOBeTKAVE/m/OzhEravZAAAJ (visited on 2021-02-17).

[Hul+20] Andreas Hulsing, Daniel J. Bernstein, Christoph Dobraunig, Maria Eichlseder, Scott Fluhrer, Stefan-Lukas Gazdag, Panos Kampanakis, Stefan Kolbl, Tanja Lange, Martin M Lauridsen, Florian Mendel, Ruben Niederhagen, Christian Rechberger, Joost Rijneveld, Peter Schwabe, Jean-Philippe Aumasson, Bas Westerbaan, and Ward Beullens. *SPHINCS+*. Tech. rep. available at https://csrc.nist.gov/projects/post-quantum-cryptography/round-3-submissions. National Institute of Standards and Technology, 2020.

[Imd] *Imdea-Software/Verifying-Constant-Time*. IMDEA Software Institute, 2015. URL: https://github.com/imdea-software/verifying-constant-time (visited on 2020-10-08).

[Imp] *ImperialViolet - Checking That Functions Are Constant Time with Valgrind*. 2010. URL: https://www.imperialviolet.org/2010/04/01/ctgrind.html (visited on 2021-02-18).

[Imp95] R. Impagliazzo. "A Personal View of Average-Case Complexity". In: *Proceedings of Structure in Complexity Theory. Tenth Annual IEEE Conference*. Structure in Complexity Theory. Tenth Annual IEEE Conference. Minneapolis, MN, USA: IEEE Comput. Soc. Press, 1995, pp. 134–147. DOI: 10.1109/SCT.1995.514853. URL: http://ieeexplore.ieee.org/document/514853/ (visited on 2021-05-10).

[Inl] *Inline Function Specifier - Cppreference.Com*. URL: https://en.cppreference.com/w/c/language/inline (visited on 2021-03-15).

[Int] *Intel® 64 and IA-32 Architectures Software Developer's Manual, Combined Volumes: 1, 2A, 2B, 2C, 2D, 3A, 3B, 3C, 3D and 4*. 2019.

[Irv53] Irving S. Reed. *A Class of Multiple-Error-Correcting Codes and the Decoding Scheme*. 1953-10-09, p. 15. URL: https://apps.dtic.mil/sti/pdfs/AD0025814.pdf (visited on 2020-12-27).

[Jao+20] David Jao, Reza Azarderakhsh, Matthew Campagna, Craig Costello, Luca De Feo, Basil Hess, Amir Jalali, Brian Koziel, Brian LaMacchia, Patrick Longa, Michael Naehrig, Joost Renes, Vladimir Soukharev, David Urbanik, Geovandro Pereira, Koray Karabina, and Aaron Hutchinson. *SIKE*. Tech. rep. available at https://csrc.nist.gov/projects/post-quantum-cryptography/round-3-submissions. National Institute of Standards and Technology, 2020.

[Jed] *Jedisct1/Libsodium*. GitHub. URL: https://github.com/jedisct1/libsodium (visited on 2021-05-09).

[KCP16] John Kelsey, Shu-jen Change, and Ray Perlner. *SHA-3 Derived Functions: cSHAKE, KMAC, TupleHash and ParallelHash*. NIST SP 800-185. Gaithersburg, MD: National Institute of Standards and Technology, 2016-12, NIST SP 800–185. DOI: 10.6028/NIST.SP.800-185. URL: https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-185.pdf (visited on 2021-05-14).

[KI03] Kazukuni Kobara and Hideki Imai. "On the One-Wayness against Chosen-Plaintext Attacks of the Loidreau's Modified McEliece PKC". In: *IEEE Transactions on Information Theory* 49.12 (2003-12), pp. 3160–3168. ISSN: 1557-9654. DOI: 10.1109/TIT.2003.820016.

[Kim+14] Yoongu Kim, Ross Daly, Jeremie Kim, Chris Fallin, Ji Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. "Flipping Bits in Memory without Accessing Them: An Experimental Study of DRAM Disturbance Errors". In: *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*. 2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA). Minneapolis, MN, USA: IEEE, 2014-06, pp. 361–372. DOI: 10.1109/ISCA.2014.6853210. URL: http://ieeexplore.ieee.org/document/6853210/ (visited on 2021-03-23).

[KJJ99] Paul C. Kocher, Joshua Jaffe, and Benjamin Jun. "Differential Power Analysis". In: *Advances in Cryptology – CRYPTO'99*. Ed. by Michael J. Wiener. Vol. 1666. Lecture Notes in Computer Science. Santa Barbara, CA, USA: Springer, Heidelberg, Germany, 1999, pp. 388–397. DOI: 10.1007/3-540-48405-1_25.

[KL14] Jonathan Katz and Yehuda Lindell. *Introduction to Modern Cryptography, Second Edition*. 2nd ed. Chapman & Hall/CRC, 2014. 603 pp. ISBN: 978-1-4665-7026-9.

[Knu97] Donald E Knuth. *The Art of Computer Programming*. Reading, Mass.; Harlow: Addison-Wesley., 1997. ISBN: 978-0-201-89684-8.

[Koc96] Paul C. Kocher. "Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems". In: *Advances in Cryptology – CRYPTO'96*. Ed. by Neal Koblitz. Vol. 1109. Lecture Notes in Computer Science. Santa

Barbara, CA, USA: Springer, Heidelberg, Germany, 1996, pp. 104–113. DOI: 10.1007/3-540-68697-5_9.

[Lan10] Adam Langley. *Agl/Ctgrind*. 2010. URL: https://github.com/agl/ctgrind (visited on 2020-10-08).

[Lip+20] Moritz Lipp, Michael Schwarz, Lukas Raab, Lukas Lamster, Misiker Tadesse Aga, Clémentine Maurice, and Daniel Gruss. "Nethammer: Inducing Rowhammer Faults through Network Requests". In: *2020 IEEE European Symposium on Security and Privacy Workshops (EuroS PW)*. 2020 IEEE European Symposium on Security and Privacy Workshops (EuroS PW). 2020-09, pp. 710–719. DOI: 10.1109/EuroSPW51379.2020.00102.

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

[Lyu+20] Vadim Lyubashevsky, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Peter Schwabe, Gregor Seiler, Damien Stehlé, and Shi Bai. *CRYSTALS-DILITHIUM*. Tech. rep. available at https://csrc.nist.gov/projects/post-quantum-cryptography/round-3-submissions. National Institute of Standards and Technology, 2020.

[Lö15] Carl Löndahl. "Some Notes on Code-Based Cryptography". 2015. URL: https://lup.lub.lu.se/search/ws/files/6280818/4934007.pdf (visited on 2021-02-17).

[McS13] Frank McSherry. "Differential Dataflow". In: (2013). URL: http://cidrdb.org/cidr2013/Papers/CIDR13_Paper111.pdf.

[Mea] *Measurements of Key-Encapsulation Mechanisms, Indexed by Machine*. URL: https://bench.cr.yp.to/results-kem.html (visited on 2021-04-16).

[Mel+20] Carlos Aguilar Melchor, Jean-Christophe Deneuville, Nicolas Aragon, Philippe Gaborit, Edoardo Persichetti, Slim Bettaieb, Jean-Marc Robert, Loïc Bidoux, Pascal Véron, Olivier Blazy, Gilles Zémor, and Jurjen Bos. *Hamming Quasi-Cyclic (HQC) Second Round Version Updated Version 2020-05-29*. 2020.

[Meu12] Alexander Meurer. "A Coding-Theoretic Approach to Cryptanalysis". 2012-11. URL: https://www.crypto.ruhr-uni-bochum.de/imperia/md/content/diss.pdf (visited on 2020-11-28).

[MS77] Florence Jessie MacWilliams and Neil James Alexander Sloane. *The Theory of Error Correcting Codes*. North-Holland Mathematical Library ; v. 16. Amsterdam ; New York : New York: North-Holland Pub. Co. ; sole distributors for the U.S.A. and Canada, Elsevier/North-Holland, 1977. 2 pp. ISBN: 978-0-444-85009-6.

[Mur+13] Derek G. Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. "Naiad: A Timely Dataflow System". In: *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. SOSP '13: ACM SIGOPS 24th Symposium on Operating Systems Principles. Farminton Pennsylvania: ACM, 2013-11-03, pp. 439–455. DOI: 10.1145/2517349.2522738.

[Mur+20] Kit Murdock, David Oswald, Flavio D. Garcia, Jo Van Bulck, Daniel Gruss, and Frank Piessens. "Plundervolt: Software-Based Fault Injection Attacks against Intel SGX". In: *2020 IEEE Symposium on Security and Privacy (SP)*. 2020 IEEE Symposium on Security and Privacy (SP). San Francisco, CA, USA: IEEE, 2020-05, pp. 1466–1482. DOI: 10.1109/SP40000.2020.00057. URL: https://ieeexplore.ieee.org/document/9152636/ (visited on 2021-03-23).

[MvV97] Alfred J. Menezes, Paul C. van Oorschot, and Scott A. Vanstone. *Handbook of Applied Cryptography*. The CRC Press series on discrete mathematics and its applications. CRC Press, 1997, pp. xxviii + 780. ISBN: 0-8493-8523-7.

[Nae+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*. Tech. rep. available at https://csrc.nist.gov/projects/post-quantum-cryptography/round-3-submissions. National Institute of Standards and Technology, 2020.

[NJW18] Alexander Nilsson, Thomas Johansson, and Paul Stankovski Wagner. "Error Amplification in Code-based Cryptography". In: *IACR Transactions on Cryptographic Hardware and Embedded Systems* 2019.1 (2018). https://tches.iacr.org/index.php/TCHES/article/view/7340, pp. 238–258. ISSN: 2569-2925. DOI: 10.13154/tches.v2019.i1.238-258.

[Par] *ParBridge Using Too Few Threads Concurrently · Issue #848 · Rayon-Rs/Rayon*. GitHub. 2021. URL: https://github.com/rayon-rs/rayon/issues/848 (visited on 2021-04-24).

[pon] poncho. *Cryptographic Random Number on Interval $[0,n]$ from Bytes*. Cryptography Stack Exchange. URL: https://crypto.stackexchange.com/questions/39165/cryptographic-random-number-on-interval-0-n-from-bytes (visited on 2021-03-23).

[Pqc] *PQClean/PQClean*. PQClean, 2021. URL: https://github.com/PQClean/PQClean (visited on 2020-11-29).

[PT19] Thales Bandiera Paiva and Routo Terada. "A Timing Attack on the HQC Encryption Scheme". In: *SAC 2019: 26th Annual International Workshop on Selected Areas in Cryptography*. Ed. by Kenneth G. Paterson and Douglas Stebila. Vol. 11959. Lecture Notes in Computer Science. Waterloo, ON,

Canada: Springer, Heidelberg, Germany, 2019, pp. 551–573. DOI: 10.1007/978-3-030-38471-5_22.

[Ray] *Rayon-Rs/Rayon*. rayon-rs, 2021. URL: https://github.com/rayon-rs/rayon (visited on 2021-02-27).

[Rep20] Oscar Reparaz. *Dude, Is My Code Constant Time?* 2020-02-18. URL: https://web.archive.org/web/20200218162023/https://www.reparaz.net/oscar/misc/dudect.html (visited on 2021-02-17).

[Rob06] Robert H. Morelos-Zaragoza. *The Art of Error Correcting Coding*. 2nd ed. Wiley, 2006-10. ISBN: 978-0-470-03570-2.

[Rob78] Robert J. McEliece. *A Public-Key Cryptosystem Based On Algebraic Coding Theory*. 1978. URL: https://ipnpr.jpl.nasa.gov/progress_report2/42-44/44N.PDF (visited on 2021-02-17).

[RQPA16] Bruno Rodrigues, Fernando Magno Quintão Pereira, and Diego F. Aranha. "Sparse Representation of Implicit Flows with Applications to Side-Channel Detection". In: *Proceedings of the 25th International Conference on Compiler Construction - CC 2016*. The 25th International Conference. Barcelona, Spain: ACM Press, 2016, pp. 110–120.

[Sch+20] Peter Schwabe, Roberto Avanzi, Joppe Bos, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, John M. Schanck, Gregor Seiler, and Damien Stehlé. *CRYSTALS-KYBER*. Tech. rep. available at https://csrc.nist.gov/projects/post-quantum-cryptography/round-3-submissions. National Institute of Standards and Technology, 2020.

[Sha82] Adi Shamir. "A Polynomial Time Algorithm for Breaking the Basic Merkle-Hellman Cryptosystem". In: *Advances in Cryptology – CRYPTO'82*. Ed. by David Chaum, Ronald L. Rivest, and Alan T. Sherman. Santa Barbara, CA, USA: Plenum Press, New York, USA, 1982, pp. 279–288.

[Sho04] Victor Shoup. *Sequences of games: a tool for taming complexity in security proofs*. Cryptology ePrint Archive, Report 2004/332. https://eprint.iacr.org/2004/332. 2004.

[Sho94] Peter W. Shor. "Algorithms for Quantum Computation: Discrete Logarithms and Factoring". In: *Proceedings 35th Annual Symposium on Foundations of Computer Science*. Proceedings 35th Annual Symposium on Foundations of Computer Science. 1994-11, pp. 124–134. DOI: 10.1109/SFCS.1994.365700.

[Sta19] François-Xavier Standaert. "How (Not) to Use Welch's T-Test in Side-Channel Security Evaluations". In: *Smart Card Research and Advanced Applications*. Ed. by Begül Bilgin and Jean-Bernard Fischer. Vol. 11389. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2019, pp. 65–79.

[Ste+20] William Stein, Frédéric Chapoton, Jeroen Demeyer, Matthias Köppe, Daniel Krenn, Julian Rüth, Nathanncohen, Volker Braun, Vincent Delecroix, Dcoudert, John H. Palmieri, Dima Pasechnik, E. M. Bray, Cheuberg, Ralf Stephan, Robert Bradshaw, Nicolas M. Thiéry, Darij Grinberg, Jm58660, Mike Hansen, Eric Gourgoulhon, Simon King, Peter Bruin, Benjamin Hackl, Roed314, Kliem, John Cremona, Martin Rubey, Marc Mezzarobba, and Yi. *Sagemath-/Sage: 9.1*. Version 9.1. Zenodo, 2020-06-24.

[SZ05] Jeff Stuckman and Guo-Qiang Zhang. *Mastermind Is NP-Complete*. 2005-12-12. arXiv: cs/0512049. URL: http://arxiv.org/abs/cs/0512049 (visited on 2021-04-10).

[Tima] *TIMECOP*. URL: https://post-apocalyptic-crypto.org/timecop/index.html#results (visited on 2021-02-17).

[Timb] *TimelyDataflow/Differential-Dataflow*. Timely Dataflow, 2021. URL: https://github.com/TimelyDataflow/differential-dataflow (visited on 2021-02-26).

[TW19] Mehdi Tibouchi and Alexandre Wallet. *One Bit is All It Takes: A Devastating Timing Attack on BLISS's Non-Constant Time Sign Flips*. Cryptology ePrint Archive, Report 2019/898. https://eprint.iacr.org/2019/898. 2019.

[van+20] Tom van Goethem, Christina Pöpper, Wouter Joosen, and Mathy Vanhoef. "Timeless Timing Attacks: Exploiting Concurrency to Leak Secrets over Remote Connections". In: *USENIX Security 2020: 29th USENIX Security Symposium*. Ed. by Srdjan Capkun and Franziska Roesner. USENIX Association, 2020, pp. 1985–2002.

[VP06] Venkat Guruswami and Prasad Raghavendra. *Lecture 8: Reed Muller Codes - CSE 533: Error-Correcting Codes*. 2006. URL: https://courses.cs.washington.edu/courses/cse533/06au/lecnotes/lecture8.pdf (visited on 2021-08-02).

[Wel47] Bernard Lewis Welch. "The Generalization of 'Student's' Problem When Several Different Population Varlances Are Involved". In: *Biometrika* 34.1-2 (1947), pp. 28–35. ISSN: 0006-3444, 1464-3510. DOI: 10.1093/biomet/34.1-2.28. URL: https://academic.oup.com/biomet/article-lookup/doi/10.1093/biomet/34.1-2.28 (visited on 2021-05-12).

[WSN18] Wen Wang, Jakub Szefer, and Ruben Niederhagen. "FPGA-Based Niederreiter Cryptosystem Using Binary Goppa Codes". In: *Post-Quantum Cryptography - 9th International Conference, PQCrypto 2018*. Ed. by Tanja Lange and Rainer Steinwandt. Fort Lauderdale, Florida, United States: Springer, Heidelberg, Germany, 2018, pp. 77–98. DOI: 10.1007/978-3-319-79063-3_4.

[WT+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. https://eprint.iacr.org/2019/909. 2019.

[Zin+17] Jean Karim Zinzindohoué, Karthikeyan Bhargavan, Jonathan Protzenko, and Benjamin Beurdouche. "HACL*: A Verified Modern Cryptographic Library". In: *ACM CCS 2017: 24th Conference on Computer and Communications Security*. Ed. by Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu. Dallas, TX, USA: ACM Press, 2017, pp. 1789–1806. DOI: 10.1145/3133956.3134043.