

Politecnico di Torino

Politecnico di Torino

Master's Degree in Cybersecurity
LM32 - Computer Engineering

Academic Year 2025/2026
September 2026

Code-Based Post-Quantum Cryptography: Analysis and Comparison of Classic McEliece and HQC

Supervisor:
Prof. Nadir Murru

Candidate:
Elena Cattelino

Abstract

The advent of large-scale quantum computers threatens the security of classical public-key cryptography, as Shor's algorithm efficiently solves the integer factorization and discrete logarithm problems underlying RSA and elliptic-curve schemes. Code-based cryptography, whose security relies on the conjectured hardness of decoding random linear codes, is among the well-studied families of post-quantum alternatives.

This thesis provides a comprehensive treatment of code-based public-key encryption and key encapsulation, combining theoretical foundations with practical evaluation. After introducing the necessary background in coding theory and in formal cryptographic security models, the thesis examines in detail two NIST-relevant code-based frameworks: the Classic McEliece cryptosystem, based on hiding the structure of a binary Goppa code, and HQC (Hamming Quasi-Cyclic), which instead exposes a public, efficiently decodable code and relies on the hardness of quasi-cyclic syndrome decoding.

Building on this theoretical foundation, the thesis presents an experimental benchmark of reference PQClean implementations of Classic McEliece and HQC across all NIST security levels, comparing key and ciphertext sizes, execution time, CPU-cycle counts, memory footprint, and measurement stability under a controlled experimental methodology designed to isolate real algorithmic effects from environmental noise.

Finally, the thesis addresses the gap between algorithmic and implementation security, reviewing two key-recovery side-channel attacks from the literature, one exploiting FFT-based decoding leakage in Classic McEliece, and one exploiting Reed–Muller decoding leakage in the RMRS instantiation of HQC. This analysis is complemented by an original software level side-channel investigation of the PQClean implementations, assessing their practical susceptibility to such leakage.

Taken together, these contributions offer both a theoretical and an empirical perspective on the trade-offs between security, efficiency, and physical robustness in code-based post-quantum key encapsulation mechanisms.

Summary

The advent of large-scale quantum computers threatens the security of classical public-key cryptography, since Shor's algorithm efficiently solves the integer factorization and discrete logarithm problems underlying RSA and elliptic-curve schemes. Code-based cryptography, whose security relies on the conjectured hardness of decoding random linear codes, is one of the best-studied families of post-quantum alternatives.

This thesis provides a comprehensive treatment of code-based public-key encryption (PKE) and key encapsulation mechanisms (KEM), combining theory with practical evaluation. It focuses on two NIST-relevant frameworks. Classic McEliece hides the structure of a binary Goppa code behind a scrambling transformation, so that breaking it amounts to solving a generic-looking syndrome decoding instance. HQC (Hamming Quasi-Cyclic) follows the opposite philosophy: it keeps its code public and efficiently decodable, and hides the message under an error term too heavy for that code's own decoder, relying on the hardness of quasi-cyclic syndrome decoding over $\mathscr{R} = \mathbb{F}_2[x]/(x^n - 1)$.
The objectives are: (i) to present the theoretical background of the two schemes, (ii) to compare their performance across all NIST security levels, and (iii) to assess the gap between algorithmic and implementation security through a side-channel analysis.

Methodologies

Theoretical part. The thesis introduces the necessary background in coding theory, the formal definitions of PKE and KEM with the security models up to IND-CCA, the main computational problems in coding theory, and the Fujisaki–Okamoto

transformation. It then describes the McEliece and HQC cryptosystems, their KEM instantiations through KeyGen, Encaps and Decaps, and their parameter sets.

Benchmark. The comparison uses the PQClean reference implementations of all Classic McEliece variants, standard and "f" (e.g. mceliece348864, mceliece6960119f), and of hqc-128, hqc-192, hqc-256, at NIST Levels 1, 3 and 5. For each of KeyGen, Encaps and Decaps, the harness performs 10000 repetitions and records wall-clock time, CPU cycles, key and ciphertext sizes, and peak memory usage. Noise is limited by pinning the process to one core, raising its priority, and running warm-ups. Results are summarized through means, standard deviations, coefficients of variation and 95% confidence intervals.

Side-channel attacks from the literature. After introducing side-channel attacks and the TVLA framework for leakage detection, the thesis analyzes in detail two profiled key-recovery attacks from the literature, which exploit weaknesses of the implementation rather than of the underlying mathematical problems.
The first, against Classic McEliece, targets the FFT-based polynomial evaluation step of the decoder: although its control flow is constant, its data flow depends on the error locator, and this dependence is visible in the power consumption. By submitting chosen ciphertexts, a classifier trained on power traces recovers the secret support of the Goppa code, from which the remaining part of the private key is derived.
The second, against the RMRS instantiation of HQC, targets the Reed–Muller decoder: a specially chosen ciphertext forces the device to decode the secret key itself, and electromagnetic traces of the Hadamard-transform step reveal the structure of the secret polynomial, which allows its full recovery.

Original software-level analysis. On the same PQClean builds, the thesis asks whether any leakage is visible from the plain CPU-cycle count of Decaps, with no probe and no classifier. It applies the Welch test to 10000 samples per class, after trimming outliers at the 1st and 99th percentiles, in two experiments: fixed vs. random ciphertext, and valid vs. corrupted ciphertext (one flipped bit, or half of the bits flipped).

Results and Conclusions

Benchmark. At every security level HQC offers smaller keys, a smaller total size of key plus ciphertext, lower total time, lower memory usage and better stability, while Classic McEliece offers faster Encaps and smaller ciphertexts. This follows from the algorithms: McEliece's Encaps is a single matrix–vector product, whereas KeyGen requires Gaussian elimination on a matrix growing with n and Decaps a full syndrome decoding; HQC spreads a moderate polynomial-ring cost over Encaps and Decaps. The absolute time gap grows from about 110 ms at Level 1 to 260–350 ms at Level 5. Within the McEliece family, the error weight t matters more than the code length n: mceliece6960119 ($n = 6960$, $t = 119$) outperforms mceliece6688128 ($n = 6688$, $t = 128$) on almost every metric. The "f" variants accept a semi-systematic public key, remove most KeyGen restarts, roughly halve its mean time and lower its coefficient of variation from over 50% to about 2% at no measurable cost. HQC is thus the sensible default for interactive or bandwidth-constrained scenarios, McEliece for keys generated once and reused many times.

Side-channel analysis. Classic McEliece shows no dependence of the Decaps cycle count on the ciphertext in any experiment and for any of the five parameter sets (maximum $|t_{\text{Welch}}| = 1.851$), consistently with its branch-free decoding pipeline. HQC behaves in the same way, except when half of the ciphertext bits are corrupted: a highly significant difference appears, with $t_{\text{Welch}} = -10.5$ for hqc-128 and $t_{\text{Welch}} = -6.6$ for hqc-192, while hqc-256 stays just below the threshold ($t_{\text{Welch}} = -1.9$). Since HQC's code is also branch-free, the effect is attributed to microarchitectural phenomena such as data-dependent instruction latency. The absolute difference between the class means is roughly constant across levels ($\approx 35\,800$–$48\,100$ cycles) while the noise grows with the cost of Decaps, so the loss of significance at Level 5 reflects an insufficient sample size, not a vanishing leak. A single corrupted bit is undetectable for both schemes. Together with the literature, this shows that a branch-free implementation is not sufficient to exclude leakage, while the software-level test is a cheap first check for implementers.

Limitations and future work. All experiments used a single platform and unmodified reference code. Future work should use different hardware and larger samples, investigate the microarchitectural origin of the HQC signal, and extend the comparison to optimized or hardened implementations and to other code-based candidates.

Contents

1 Preliminaries **1**

1.1 Notation . . . 1

1.2 Coding Theory . . . 2

1.2.1 Linear Codes . . . 3

1.2.2 Matrix Codes . . . 7

1.3 Cryptography . . . 18

1.3.1 Public-Key Encryption (PKE) . . . 18

1.3.2 Key Encapsulation Mechanism (KEM) . . . 20

1.4 Security Considerations . . . 22

1.4.1 Complexity Classes . . . 22

1.4.2 Computational Problems in Coding Theory . . . 24

1.4.3 Formal Security Models . . . 28

1.4.4 Fujisaki-Okamoto Transformation . . . 32

2 Code-Based Public-Key Encryption Frameworks **38**

2.1 Post-Quantum Cryptographic Frameworks . . . 38

2.2 McEliece Cryptosystem . . . 43

2.2.1 McEliece Framework for Key Encapsulation Mechanisms . . . 47

2.2.2 Classic McEliece Parameters . . . 49

2.3 HQC Cryptosystem . . . 50

2.3.1 HQC Framework for Key Encapsulation Mechanisms 55
2.3.2 HQC Parameters 56

3 Benchmark of McEliece and HQC Key Encapsulation Mechanisms 58
3.1 Motivations and Objectives of the Benchmark 58
3.2 Benchmark Environment and Experimental Methodology 59
3.2.1 Reference Implementations and Parameter Sets 59
3.2.2 Control of Environmental Noise 61
3.2.3 Timing and Cycle-Counting Instrumentation 62
3.2.4 Statistical Treatment of the Measurements 63
3.2.5 Correctness Verification 63
3.3 Description of the Test Suite and of the Code Structure 64
3.3.1 Helper and Infrastructure Functions 64
3.3.2 Process-Isolation Layer 66
3.3.3 Benchmark Functions 68
3.3.4 Reporting Functions 70
3.4 Results 71
3.4.1 NIST Security Level 1 72
3.4.2 NIST Security Level 3 74
3.4.3 NIST Security Level 5 76
3.5 Conclusions 82
3.5.1 McEliece vs HQC at Each Level 82
3.5.2 McEliece variants at Level 5 83
3.5.3 McEliece Standard vs “f” Variants 84
3.5.4 Conclusions 84

4 Analysis of Side Channel Attacks on McEliece and HQC Key Encapsulation Mechanism 86

4.1 Introduction to Side-Channel Attacks 86
4.1.1 Leakage Detection with TVLA and the Welch t-test 87
4.1.2 Case Studies 87
4.2 A Key-Recovery Side-Channel Attack on Classic McEliece 88
4.2.1 Introduction 88
4.2.2 The Attack Algorithm 90
4.2.3 Experimental Results 92
4.3 A New Key Recovery Side-Channel Attack on HQC with Chosen Ciphertext 93
4.3.1 The RMRS Instantiation of HQC 93
4.3.2 The Threat Model 96
4.3.3 The Attack Algorithm 99
4.3.4 Experimental Results 100

5 Side Chanel Analysis of McEliece and HQC Key Encapsulation Mechanisms 103

5.1 Motivations and Objectives of the Analysis 103
5.2 Analysis Environment and Experimental Methodology 104
5.2.1 Reuse of the Environmental Controls 104
5.2.2 Choice of Timing Instrumentation 105
5.2.3 Experiment 1: Fixed vs. Random Ciphertext 105
5.2.4 Experiment 2: Valid vs. Corrupted Ciphertext 105
5.2.5 Statistical Treatment of the Measurements 106
5.3 Description of the Test Suite and of the Code Structure 107
5.3.1 Reused Helper and Infrastructure Functions 107
5.3.2 Ciphertext Corruption 107
5.3.3 Measurement Routines 108

5.3.4 Main Function 110
5.3.5 Offline Analysis Script 111
5.4 Results 112
5.4.1 Classic McEliece 112
5.4.2 HQC 122
5.5 Conclusions 128
5.5.1 Classic McEliece 128
5.5.2 HQC 129
5.5.3 Conclusions 131

6 Conclusions and Future Work **132**

References **134**

Chapter 1

Preliminaries

1.1 Notation

- Scalars are denoted by lowercase italic letters (e.g., a, b, α);
- Vectors are denoted by bold lowercase letters (e.g., $\mathbf{v}, \mathbf{x}, \mathbf{e}$);
- Matrices are denoted by uppercase letters (e.g., G, H, A);
- Sets are denoted by calligraphic letters (e.g., $\mathscr{S}, \mathscr{F}$);
- Finite fields are denoted by blackboard bold letters (e.g., $\mathbb{F}_q$);
- The vector space of length-n vectors over $\mathbb{F}_q$ is denoted by $\mathbb{F}_q^n$;
- Linear codes are denoted by $\mathscr{C} \subseteq \mathbb{F}_q^n$;
- The dimension of a linear code $\mathscr{C}$ is denoted by k, and its length by n;
- Generator and parity-check matrices are denoted by $G \in \mathbb{F}_q^{k \times n}$ and $H \in \mathbb{F}_q^{(n-k) \times n}$, respectively;
- The Hamming weight of a vector $\mathbf{v}$ is denoted by $\mathtt{wt}_H(\mathbf{v})$;
- The Hamming distance between two vectors $\mathbf{u}, \mathbf{v}$ is denoted by $d(\mathbf{u}, \mathbf{v})$;
- Polynomials are denoted by lowercase or uppercase letters with an argument (e.g. $g(x)$, $\Gamma(x)$);

- Polynomial rings over finite fields are denoted by $\mathscr{R} = \mathbb{F}_q[x]/(x^n - 1)$;
- Random sampling is denoted by $x \xleftarrow{\$} \mathscr{X}$.

1.2 Coding Theory

An introduction to algebraic coding theory is necessary to present the cryptographic frameworks. The theoretical background on algebraic coding theory presented in this chapter is inspired by the work of Weger, Gassner, and Rosenthal [1], to which the reader is referred for further details.

In modern digital communication systems, reliable transmission of information over noisy channels is a fundamental challenge. Coding theory addresses this problem by introducing structured redundancy into transmitted data, enabling the detection and correction of errors caused by channel noise, interference, or other impairments.

Given a finite alphabet $\mathscr{F}$, consider a message $\mathbf{m} \in \mathscr{F}^k$ that needs to be transmitted. In order to detect and correct errors during the transmission, $\mathbf{m}$ is encoded into a new vector $\mathbf{c} \in \mathscr{F}^n$, with $n > k$, adding redundancy. The vector $\mathbf{c}$ is called a codeword.
A code $\mathscr{C}$ is the set of all codewords, and it is a structured subset of $\mathscr{F}^n$ containing $|\mathscr{F}|^k$ elements.

The algebraic structure of $\mathscr{C}$ allows the receiver to detect and correct transmission errors. In a typical communication system:

- The sender encodes the message $\mathbf{m}$ into a codeword $\mathbf{c}$;
- The codeword is transmitted over a noisy channel, potentially becoming corrupted;
- The receiver uses the redundancy in $\mathscr{C}$ to identify and correct errors, recovering the original $\mathbf{m}$.

This structured use of redundancy forms the core of error-control coding, improving data integrity and system efficiency without requiring retransmission.

1.2.1 Linear Codes

In classical coding theory the finite field of q elements $\mathbb{F}_q$ is used as finite alphabet $\mathscr{F}$. In this context we can define a linear code.

Definition 1 (Linear Code). Let $1 \leq k \leq n$ be integers. Then a $[n,k,d]$-code $\mathscr{C}$ is said linear if it is a linear subspace of $\mathbb{F}_q^n$ over $\mathbb{F}_q$.

The notation $[n,k,d]$ denotes a code of length n, dimension k and minimum distance d.
The length n represents the number of entries (which are elements of $\mathbb{F}_q$, e.g., bits if $q=2$) of each codeword, while k is the number of entries carrying on the information contained in a codeword. Consequently, the difference $n-k$ is called the redundancy of the code, and $R=k/n$ is called the rate of the code. The rate measures the fraction of entries in a codeword that carry actual information.
If the dimension of the linear code is k, then the basis contains k linearly independent vectors, and the codewords are all linear combinations of those vectors. Thus, the size of the code is $|\mathscr{C}| = q^k$.
In order to describe the parameter d, a distance among the elements of the whole space $\mathbb{F}_q^n$ is defined. Usually the Hamming distance is used.

Definition 2 (Hamming Weigh). Let n be a positive integer. Given $\mathbf{x} \in \mathbb{F}_q^n$, the Hamming weight is the number of non-zero coordinates of $\mathbf{x}$, i.e,

$$\mathsf{wt}_H(\mathbf{x}) = |\{i \in \{1,\ldots,n\} \mid x_i \neq 0\}|.$$

Definition 3 (Hamming Distance). Let n be a positive integer. Given $\mathbf{x},\mathbf{y} \in \mathbb{F}_q^n$, the Hamming distance $d_H(\mathbf{x},\mathbf{y})$ is the number of coordinates in which $\mathbf{x}$ and $\mathbf{y}$ differ, i.e.,

$$d_H(\mathbf{x},\mathbf{y}) = |\{i \in \{1,\ldots,n\} \mid x_i \neq y_i\}|.$$

Definition 4 (Minimum Distance). Let $\mathscr{C}$ be a code over $\mathbb{F}_q$. The minimun (Hamming) distance of $\mathscr{C}$ is denoted by $d_H(\mathscr{C})$ and given by

$$d_H(\mathscr{C}) = \min\{d_H(\mathbf{x},\mathbf{y}) \mid \mathbf{x},\mathbf{y} \in \mathscr{C},\ \mathbf{x} \neq \mathbf{y}\}.$$

It can be observed that in a linear code the null vector $\mathbf{0} = (0 \dots 0)$ is always a codeword. Moreover, for any two codewords $\mathbf{c_1}, \mathbf{c_2} \in \mathscr{C}$ the following equality holds:

$$d_H(\mathbf{c_1}, \mathbf{c_2}) = d_H(\mathbf{c_1} - \mathbf{c_2}, \mathbf{0}).$$

As a consequence, the minimum distance in a linear code is equal to the minimum Hamming weight among all non-zero codewords of the code:

$$d_H = \min_{\mathbf{x} \in \mathscr{C}, \mathbf{x} \neq \mathbf{0}} \mathsf{wt}_H(\mathbf{x}).$$

From this point onward, the minimun distance of a code will always be considered as a minimum Hamming distance, so the notation d_H will be simplified to d. Moreover, in the notation $[n,k,d]$-code the minimum distance parameter can be omitted.

There exists an inherent trade-off between a code's rate and its error-correction capability (governed by d). This trade-off is demonstrated by several fundamental bounds in coding theory, like the Singleton bound.

Theorem 1 (Singleton Bound). For any linear $[n,k,d]$-code over $\mathbb{F}_q$, the parameters satisfy

$$d \leq n - k + 1.$$

A code achieving the Singleton Bound ($d = n - k + 1$) is called Maximum Distance Separable (MDS). Generalized Reed-Solomon (GRS) codes are a primary example of MDS codes.

Error Detection and Correction

Given the minimum distance d of a linear code $\mathscr{C}$, it is possible to determine the maximum number of errors that the system can detect and correct.

Definition 5 (Sphere). Let r be a positive integer. Given $\mathbf{x} \in \mathbb{F}_q^n$, the sphere of radius r and center $\mathbf{x}$ is defined as

$$S_r(\mathbf{x}) = \left\{ \mathbf{x}' \in \mathbb{F}_q^n \,:\, d(\mathbf{x}, \mathbf{x}') \leq r \right\}.$$

Then, given a linear code $\mathscr{C}$ and an integer h_1, $\mathscr{C}$ is able to detect h_1 errors if and only if for all $\mathbf{c} \in \mathscr{C}$ the following condition holds:

$$S_{h_1}(\mathbf{c}) \cap \mathscr{C} = \{\mathbf{c}\}.$$

Moreover, given a linear code $\mathscr{C}$ and an integer h_2, $\mathscr{C}$ is able to correct h_2 errors if and only if for all distinct $\mathbf{c_1}, \mathbf{c_2} \in \mathscr{C}$ the following condition holds:

$$S_{h_2}(\mathbf{c_1}) \cap S_{h_2}(\mathbf{c_2}) = \emptyset.$$

From the definitions above, the following theorem can be derived.

Theorem 2. Given a linear code $\mathscr{C}$ with minimum distance d, then $\mathscr{C}$ is able to detect $d-1$ errors and correct $\lfloor \frac{d-1}{2} \rfloor$.

Generator Matrix

Linear codes are vector subspaces of $\mathbb{F}_q^n$ and all their elements can be described in terms of a basis. This basis can be represented in a matrix form, called generator matrix.

Definition 6 (Generator Matrix). Let $k \leq n$ be positive integers and let $\mathscr{C}$ be an $[n,k]$-linear code over $\mathbb{F}_q$. Then, a matrix $G \in \mathbb{F}_q^{k \times n}$ is called a generator matrix of $\mathscr{C}$ if

$$\mathscr{C} = \left\{ \mathbf{x}G \mid \mathbf{x} \in \mathbb{F}_q^k \right\},$$

that is, the rows of G form a basis of $\mathscr{C}$.

In other words, a generator matrix is a $k \times n$ matrix whose rows form a basis of vector space $\mathscr{C}$ over $\mathbb{F}_q$.

If the first k columns of the generator matrix are linearly independent, then the linear code can be described by a systematic generator matrix. This can be obtained by multiplying G on the left by an invertible $k \times k$ matrix U and on the right by an $n \times n$ permutation matrix P such that

$$UGP = (I_k|A),$$

where I_k is a $k \times k$ identity matrix and $A \in \mathbb{F}_q^{k\times(n-k)}$.
The use of a systematic generator matrix simplifies the recovery of the information word from a codeword. Indeed, given an information word $\mathbf{m} = (m_1, \ldots, m_k) \in \mathbb{F}_q^k$, we have $\mathbf{m}G = \mathbf{c} = (m_1, \ldots, m_k, \ldots) \in \mathbb{F}_q^n$, showing that the original information word can be obtained simply by dropping the last $n-k$ symbols of the codeword.

Parity Check Matrix

A linear code can also be represented through a parity check matrix H.

Definition 7 (Parity Check Matrix). Let $k \leq n$ be positive integers and let $\mathscr{C}$ be an $[n,k]$-linear code over $\mathbb{F}_q$. Then, a matrix $H \in \mathbb{F}_q^{(n-k)\times n}$ is called a parity check matrix of $\mathscr{C}$, if

$$\mathscr{C} = \left\{ \mathbf{y} \in \mathbb{F}_q^n \mid H\mathbf{y}^\top = \mathbf{0} \right\}.$$

The parity check matrix provides an efficient way to verify whether a received word is a codeword.

For $\mathbf{x}, \mathbf{y} \in \mathbb{F}_q^n$ let us denote by $\langle \mathbf{x}, \mathbf{y} \rangle$ the standard inner product, i.e.,

$$\langle \mathbf{x}, \mathbf{y} \rangle = \sum_{i=1}^{n} x_i y_i.$$

Then, we can define the dual of an $[n,k]$-linear code $\mathscr{C}$ over $\mathbb{F}_q$ as the orthogonal space of $\mathscr{C}$.

Definition 8 (Dual Code). Let $k \leq n$ be positive integers and let $\mathscr{C}$ be an $[n,k]$-linear code over $\mathbb{F}_q$. The dual code $\mathscr{C}^\perp$ is an $[n,n-k]$-linear code over $\mathbb{F}_q$, defined as

$$\mathscr{C}^\perp = \{\mathbf{x} \in \mathbb{F}_q^n : \langle \mathbf{x}, \mathbf{y} \rangle = 0, \forall \mathbf{y} \in \mathscr{C}\}.$$

The generator matrix of the dual code $\mathscr{C}^\perp$ corresponds to the parity check matrix of $\mathscr{C}$.
Furthermore, if the generator matrix of a code is in systematic form, the corresponding parity check matrix can be immediately derived as

$$G = (I_k|A) \iff H = (-A^\top, I_{n-k}).$$

Similarly, a generic parity check matrix H can be brought into a systematic form by means of a permutation matrix P' and an invertible matrix U', i.e.,

$$U'HP' = (B|I_{n-k}),$$

where $B \in \mathbb{F}_q^{(n-k)\times k}$.

Syndrome

Another fundamental property of linear codes is the linear application $s : \mathbb{F}_q^n \to \mathbb{F}_q^{n-k}$ called syndrome. More formally

Definition 9 (Syndrome). Let $\mathscr{C}$ be a linear $[n,k]$-code, let H be its parity check matrix and let $\mathbf{y} \in \mathbb{F}_q^n$ be a received codeword. A syndrome is the linear application $s : \mathbb{F}_q^n \to \mathbb{F}_q^{n-k}$ such that

$$s(\mathbf{y}) = \mathbf{y}H^\top.$$

It can be shown that the syndrome of a received word $\mathbf{y} \in \mathbb{F}_q^n$ equals to the syndrome of the error. Indeed, given a codeword $\mathbf{c} \in \mathscr{C}$, an error $\mathbf{e} \in \mathbb{F}_q^n$ and a received word $\mathbf{y} = \mathbf{c} + \mathbf{e}$, then

$$\mathbf{y}H^\top = \mathbf{c}H^\top + \mathbf{e}H^\top.$$

Since $\mathbf{c}H^\top = \mathbf{0}$ by definition of the parity check matrix, the result is

$$s(\mathbf{y}) = s(\mathbf{e}).$$

Thus, the syndrome of the received word is exactly the syndrome of the error.
The syndrome provides a way to map a received word to a quantity that depends only on the error and not on the received codeword. For this reason, syndromes are used in decoding to identify the error affecting $\mathbf{y}$.

1.2.2 Matrix Codes

Instead of considering subspaces in $\mathbb{F}_q^n$, one can also consider subspaces in $\mathbb{F}_q^{n\times m}$, which are $n \times m$ matrices over $\mathbb{F}_q$. Those subspaces are called matrix codes.

Definition 10 (Matrix Codes). An $\mathbb{F}_q$-linear subspace of $\mathbb{F}_q^{m\times n}$ is called a matrix code.

In other words the code is generated by k generator matrices $G_1,\ldots,G_k \in \mathbb{F}_q^{m\times n}$, so that every codeword has the form

$$\mathbf{C} = \lambda_1 G_1 + \cdots + \lambda_k G_k,$$

for some $\lambda_i \in \mathbb{F}_q$.

With this new notion of a codeword, a suitable Hamming metric must be introduced. As a solution, it can be either considered the number of non-zero columns or the number of non-zero entries.
For a matrix $A \in \mathbb{F}_q^{m\times n}$, it is possible denote by $\mathbf{c}_i \in \mathbb{F}_q^m$ its columns for $i \in \{1,\ldots,n\}$, by $\mathbf{r}_j \in \mathbb{F}_q^n$ its rows for $j \in \{1,\ldots,m\}$ and finally by $a_{i,j}$ its entries for $(i,j) \in \{1,\ldots,n\}\times\{1,\ldots,m\}$. Given $A \in \mathbb{F}_q^{m\times n}$, it is possible to define

$$\begin{aligned}\mathsf{wt}_{H,c}(A) &= |\{i \in \{1,\ldots,n\} \mid \mathbf{c}_i \neq \mathbf{0}\}|,\\ \mathsf{wt}_{H,v}(A) &= |\{(i,j) \in \{1,\ldots,n\}\times\{1,\ldots,m\} \mid a_{i,j} \neq 0\}|.\end{aligned}$$

Definition 11 (Vectorization). Given a matrix $A \in \mathbb{F}_q^{m\times n}$ with rows $\mathbf{a}_1,\ldots,\mathbf{a}_m \in \mathbb{F}_q^n$, the vectorization of A is defined as $\mathrm{vec}(A) = (\mathbf{a}_1,\ldots,\mathbf{a}_m) \in \mathbb{F}_q^{mn}$.

The Hamming weight of $\mathrm{vec}(A)$ coincides with the second notion of Hamming metric of matrices, i.e.,

$$\mathsf{wt}_H(\mathrm{vec}(A)) = \mathsf{wt}_{H,v}(A).$$

Another important concept is the extension map of a code, which is a function that represents elements from an extension field $\mathbb{F}_{q^m}$ into vectors over the base field $\mathbb{F}_q$.

Definition 12 (Extension Map). Let $\Gamma = \{\gamma_1,\ldots,\gamma_m\}$ be a basis of $\mathbb{F}_{q^m}$ over $\mathbb{F}_q$. That is, every element $a \in \mathbb{F}_{q^m}$ can be written as

$$a = \sum_{i=1}^{m} a_i \gamma_i,$$

with $a_i \in \mathbb{F}_q$. Then, the extension map can be defined as

$$\begin{aligned} \Gamma : \mathbb{F}_{q^m} &\to \mathbb{F}_q^m \\ a = \sum_{i=1}^{m} a_i \gamma_i &\mapsto (a_1, \ldots, a_m). \end{aligned}$$

By abuse of notation Γ can be used to denote the extension map $\Gamma : \mathbb{F}_{q^m}^n \to \mathbb{F}_q^{m \times n}$, where each entry is extended to a column.
The Hamming weight of the vector $\Gamma^{-1}(A) = \mathbf{a} \in \mathbb{F}_{q^m}^n$ coincides with the first notion of Hamming weight for matrices, i.e.,

$$\mathsf{wt}_H(\Gamma^{-1}(A)) = \mathsf{wt}_{H,c}(A).$$

The extension map can also be applied to a code itself. Let $\mathscr{C} \subseteq \mathbb{F}_{q^m}^n$ be a linear code and let Γ be a basis of $\mathbb{F}_{q^m}$ over $\mathbb{F}_q$. The code associated with Γ is given by

$$\Gamma(\mathscr{C}) = \{\Gamma(\mathbf{c}) \mid \mathbf{c} \in \mathscr{C}\}.$$

The dual code of a matrix code, requires a new inner product, which extends the previous described standard inner product. For this, recall that the trace of a matrix is the sum of the entries on its diagonal.

Definition 13 (Trace). Let $A, B \in \mathbb{F}_q^{m \times n}$, then we define their trace product as

$$\mathrm{Tr}(AB^\top).$$

Let $\mathscr{C} \subseteq \mathbb{F}_q^{m \times n}$ be a linear matrix code, then its dual code is given by

$$\mathscr{C}^\perp = \{A \in \mathbb{F}_q^{m \times n} \mid \mathrm{Tr}(AB^\top) = \mathbf{0} \text{ for all } B \in \mathscr{C}\}.$$

Generalized Reed-Solomon Codes

Generalized Reed-Solomon (GRS) codes are a fundamental class of linear codes that achieve optimal error-correction capability. They belong to the family of Maximum Distance Separable (MDS) codes, meaning they attain the Singleton bound with equality: their minimum distance satisfies $d = n - k + 1$.

Definition 14 (Generalized Reed-Solomon Code). Let $k \leq n \leq q$ be positive integers. Let $\alpha \in \mathbb{F}_q^n$ be an n-tuple of distinct elements, i.e., $\alpha = (\alpha_1, \ldots, \alpha_n)$ with $\alpha_i \neq \alpha_j$, for all $i \neq j \in \{1, \ldots, n\}$. Let $\beta \in \mathbb{F}_q^n$ be an n-tuple of nonzero elements, i.e., $\beta = (\beta_1, \ldots, \beta_n)$, with $\beta_i \neq 0$ for all $i \in \{1, \ldots, n\}$. The Generalized Reed-Solomon code of length n and dimension k, denoted by $\mathrm{GRS}_{n,k}(\alpha, \beta)$ is defined as

$$\mathrm{GRS}_{n,k}(\alpha, \beta) = \left\{ (\beta_1 f(\alpha_1), \ldots, \beta_n f(\alpha_n)) \;\middle|\; f \in \mathbb{F}_q[x],\ \deg(f) < k \right\}.$$

When the column multipliers are all set to $\beta = (1, 1, \ldots, 1)$, the GRS code reduces to a classical Reed-Solomon (RS) code, where codewords are simply evaluations of polynomials without scaling:

$$\mathrm{RS}_{n,k}(\alpha) = \mathrm{GRS}_{n,k}(\alpha, \mathbf{1}) = \left\{ (f(\alpha_1), \ldots, f(\alpha_n)) \mid f \in \mathbb{F}_q[x],\ \deg(f) < k \right\}.$$

GRS codes generalize this construction while preserving all the desirable properties of RS codes, adding more flexibility in code costruction.

Reed-Muller Codes

Reed-Muller (RM) codes are a fundamental family of binary linear codes whose codewords consist of the evaluation vectors of multivariate polynomials over $\mathbb{F}_2$. The encoding procedure maps the information bits, stored as the coefficients of a polynomial, to its vector of evaluations over all points of $\mathbb{F}_2^m$ [2].

Definition 15 (Reed-Muller Code). Consider the polynomial ring $\mathbb{F}_2[x_1, x_2, \ldots, x_m]$ in m variables. For a polynomial $f \in \mathbb{F}_2[x_1, \ldots, x_m]$ and a vector $z = (z_1, \ldots, z_m) \in \mathbb{F}_2^m$, let $\mathrm{Eval}_z(f) := f(z_1, \ldots, z_m)$, and let $\mathrm{Eval}(f) := (\mathrm{Eval}_z(f) : z \in \mathbb{F}_2^m)$ denote the evaluation vector of f over all 2^m points of $\mathbb{F}_2^m$. The r-th order Reed-Muller code $\mathrm{RM}(r, m)$ is defined as

$$\mathrm{RM}(r, m) := \left\{ \mathrm{Eval}(f) \;\middle|\; f \in \mathbb{F}_2[x_1, x_2, \ldots, x_m],\ \deg(f) \leq r \right\}.$$

Since $x_i^2 = x_i$ over $\mathbb{F}_2$, every polynomial reduces to a linear combination of multilinear monomials $x_A := \prod_{i \in A} x_i$ for $A \subseteq [m]$ with $|A| \leq r$, so $\mathrm{RM}(r, m)$ is a linear code with length $n = 2^m$, dimension $k = \sum_{i=0}^{r} \binom{m}{i}$, and minimum distance $d = 2^{m-r}$.

In particular, $\mathrm{RM}(0,m)$ is the repetition code, while $\mathrm{RM}(m,m)$ spans all of $\mathbb{F}_2^n$, since the evaluation vectors $\{\mathrm{Eval}(x_A) : A \subseteq [m]\}$ form a basis of $\mathbb{F}_2^n$ [2].

Goppa Codes

Goppa codes are an important family of linear error-correcting codes, known for their excellent asymptotic properties and efficient decoding algorithms. Algebraically, Goppa codes can be viewed as a generalization of Reed–Solomon codes and as a special case of algebraic geometry codes.

Let m be a positive integer, $n = q^m$ and $\mathbb{F}_{q^m}$ be a finite field. Let $G \in \mathbb{F}_{q^m}[x]$. Then define the quotient ring

$$S_m = \frac{\mathbb{F}_{q^m}[x]}{\langle G \rangle}.$$

Lemma 1. Let $\alpha \in \mathbb{F}_q$ be such that $G(\alpha) \neq 0$. Then $(x-\alpha)$ is invertible in S_m and

$$(x-\alpha)^{-1} = -\frac{1}{G(\alpha)} \frac{G(x)-G(\alpha)}{x-\alpha}.$$

Definition 16 (Classical Goppa Code). Let $\alpha = (\alpha_1, \ldots, \alpha_n) \in \mathbb{F}_{q^m}^n$, be such that $\alpha_i \neq \alpha_j$ for all $i \neq j \in \{1, \ldots, n\}$, and $G(\alpha_i) \neq 0$ for all $i \in \{1, \ldots, n\}$. Then we can define the classical q-ary Goppa code as

$$\Gamma(\alpha, G) = \left\{ c \in \mathbb{F}_q^n \;\middle|\; \sum_{i=1}^{n} \frac{c_i}{x-\alpha_i} = 0 \text{ in } S_m \right\}.$$

The Goppa code $\Gamma(\alpha, G)$ has minimum Hamming distance $d_H(\Gamma(\alpha, G)) \geq \deg(G) + 1$ and dimension $k \geq n - m\deg(G)$.

An important characteristic of a Goppa code is the parity check matrix, which enables fast error correction method. In order to construct it, let us define $\beta = (G(\alpha_1)^{-1}, \ldots, G(\alpha_n)^{-1})$. The parity-check matrix of $\Gamma(\alpha, G)$ is then given by the weighted Vandermonde matrix

$$H = \begin{pmatrix} \beta_1 & \cdots & \beta_n \\ \beta_1\alpha_1 & \cdots & \beta_n\alpha_n \\ \vdots & & \vdots \\ \beta_1\alpha_1^{r-1} & \cdots & \beta_n\alpha_n^{r-1} \end{pmatrix}.$$

This construction reveals a fundamental structural link between Goppa codes and the Generalized Reed–Solomon codes introduced earlier. To make this precise, we recall two auxiliary notions.

Definition 17 (Subfield Subcode). Let $\mathscr{C}$ be an $[n,k]$ linear code over $\mathbb{F}_{q^m}$. The subfield subcode of $\mathscr{C}$ over $\mathbb{F}_q$ is then defined as

$$\mathscr{C}_{\mathbb{F}_q} = \mathscr{C} \cap \mathbb{F}_q^n.$$

If $\mathscr{C}$ has minimum distance d, then $\mathscr{C}_{\mathbb{F}_q}$ has minimum distance at least d and dimension at least $n - m(n-k)$.

Definition 18 (Alternant Code). Let $\alpha \in \mathbb{F}_{q^m}^n$ be pairwise distinct and $\beta \in (\mathbb{F}_{q^m}^\times)^n$. Then the alternant code $\mathscr{A}_{m,n,k}(\alpha,\beta)$ is defined as

$$\mathscr{A}_{m,n,k}(\alpha,\beta) = \mathrm{GRS}_{m,n,k}(\alpha,\beta) \cap \mathbb{F}_q^n.$$

The alternant code $\mathscr{A}_{m,n,k}(\alpha,\beta)$ has dimension $\geq n - m(n-k)$ and minimum distance $\geq n-k+1$.
Thus, classical Goppa codes are alternant codes, i.e., subfield subcodes of particular GRS codes, where the weights β_i are the inverses of the evaluations $g(\alpha_i)$, for a polynomial g.

Patterson's Algorithm

Patterson's algorithm is a classical decoding algorithm for Generalized Reed-Solomon (GRS) codes, which allows the correction of up to $\lfloor \frac{d-1}{2} \rfloor$ errors. The algorithm is based on the computation of two auxiliary polynomials, whose roots and coefficients reveal the error locations and values.

To start, recall the structure of the parity-check matrix, the received word model and the definition of the corresponding syndrome, which constitute the input of the decoding procedure.

The parity-check matrix of a GRS code can be written as

$$H = \begin{pmatrix} 1 & 1 & \cdots & 1 \\ \alpha_1 & \alpha_2 & \cdots & \alpha_n \\ \alpha_1^2 & \alpha_2^2 & \cdots & \alpha_n^2 \\ \vdots & \vdots & \ddots & \vdots \\ \alpha_1^{d-2} & \alpha_2^{d-2} & \cdots & \alpha_n^{d-2} \end{pmatrix} \begin{pmatrix} v_1 & 0 & \cdots & 0 \\ 0 & v_2 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & v_n \end{pmatrix},$$

where $\alpha_1, \ldots, \alpha_n$ are n distinct non-zero elements of $\mathbb{F}_q$, called code locators, and $v_1, \ldots, v_n$ are non-zero elements of $\mathbb{F}_q$, called column multipliers.
The code has length n, dimension k, and minimum distance d, and for a GRS code we have $d = n - k + 1$, hence $n - k = d - 1$.

Let $\mathbf{c}$ be a transmitted codeword, $\mathbf{e}$ the error vector, and $\mathbf{y}$ the received word, so that

$$\mathbf{y} = \mathbf{c} + \mathbf{e}.$$

It can be denoted a new element J, which is the set of error locations, i.e.,

$$e_j \neq 0 \iff j \in J.$$

Identifying the set J is equivalent to locating the positions where errors occurred and thus correcting the received word. For this reason the set's cardinality is $|J| \leq \left\lfloor \frac{d-1}{2} \right\rfloor$, as the algorithm is able to correct up to this quantity of errors.

The first step of the decoding process consists in computing the syndrome associated with the received word.
The syndrome of $\mathbf{y}$ is defined as

$$H\mathbf{y}^\top = \begin{pmatrix} S_0 \\ S_1 \\ \vdots \\ S_{d-2} \end{pmatrix}.$$

For GRS codes, the syndrome components can be explicitly written as

$$S_l = \sum_{j=1}^{n} y_j v_j \alpha_j^l, \quad l = 0, 1, \ldots, d-2.$$

In the case of RS codes, the locators and multipliers satisfy

$$\alpha_j = \alpha^{j-1}, \quad j = 1, \ldots, n,$$

where $\alpha \in \mathbb{F}_{q^m}$ has multiplicative order n, and

$$v_j = \alpha^{b(j-1)}, \quad j = 1, \ldots, n,$$

for some integer b. The syndrome components then become

$$S_l = \sum_{j=1}^{n} y_j \alpha^{(j-1)(b+l)}, \quad l = 0, 1, \ldots, d-2.$$

It is convenient to describe the received word $\mathbf{y}$ using a polynomial representation:

$$y(x) = y_1 + y_2 x + \cdots + y_n x^{n-1}.$$

With this notation, the syndrome components can be written as $S_l = y(\alpha^{b+l})$, and it is possible to define the syndrome polynomial as

$$\begin{aligned} S(x) &= S_0 + S_1 x + \cdots + S_{d-2} x^{d-2} \\ &= \sum_{l=0}^{d-2} y(\alpha^{b+l}) x^l. \end{aligned}$$

Substituting the explicit expression of $y(\alpha^{b+l})$ and assuming, without loss of generality, that $\mathbf{c} = \mathbf{0}$ (so that $\mathbf{y} = \mathbf{e}$), we obtain

$$S(x) = \sum_{l=0}^{d-2} \sum_{j=1}^{n} e_j v_j (\alpha_j x)^l.$$

It can be noticed that the polynomial $\sum_{l=0}^{d-2} (\alpha_j x)^l$ is invertible modulo x^{d-1}, indeed:

$$(1 - \alpha_j x)(1 + \alpha_j x + \cdots + \alpha_j^{d-2} x^{d-2}) = 1 - \alpha_j^{d-1} x^{d-1} \equiv 1 \pmod{x^{d-1}}.$$

Therefore the final result for the syndrome polynomial is:

$$S(x) \equiv \sum_{j \in J} \frac{e_j v_j}{1 - \alpha_j x} \pmod{x^{d-1}}.$$

Two other fundamental polynomials need to be introduced. The first one is the Error Locator Polynomial (ELP), defined as

$$\Lambda(x) = \prod_{j \in J} (1 - \alpha_j x).$$

By construction $\Lambda(\alpha_j^{-1}) = 0 \iff j \in J$, hence it is sufficient to compute $\Lambda(x)$, given the corresponding received word $y(x)$, in order to find the errors. Indeed $\alpha_1, \ldots, \alpha_n$ are known.
However, $\Lambda(x)$ cannot be computed yet, as it depends on the still unknown error set J, which is also the final goal of the algorithm. For this reason, the third and last polynomial, the Error Evaluator Polynomial (EEP), is defined

$$\Gamma(x) = \sum_{j \in J} e_j v_j \prod_{\substack{m \in J \\ m \neq j}} (1 - \alpha_m x),$$

which, like $\Lambda(x)$, cannot be immediately determined.
The strategy is extracting different relations involving all polynomials. In particular, three equations can be constructed:

1. First, since $\Lambda(x)$ has roots α_j^{-1} for $j \in J$ and $\Gamma(x)$ does not, it can be derived that
$$\gcd(\Lambda(x), \Gamma(x)) = 1;$$

2. Then, remembering that $\deg \Lambda(x) = |J|$, $\deg \Gamma(x) < |J|$, and $|J| \leq \lfloor \frac{d-1}{2} \rfloor$, the following disequations can be written
$$\deg \Gamma(x) < \deg \Lambda(x) \leq \lfloor \frac{d-1}{2} \rfloor;$$

3. Lastly, the syndrome polynomial is involved in the equivalence
$$\Lambda(x) S(x) \equiv \Gamma(x) \pmod{x^{d-1}}.$$

These relations represents the key equations of Patterson's algorithm, and uniquely determine $\Lambda(x)$ and $\Gamma(x)$, allowing the computation of the error set J.
The third equation can be solved by rewriting it as a linear system in the coefficients of $\Lambda(x)$ and $\Gamma(x)$ and solved, for example, by Gaussian elimination. The remaining

two relations guarantee the uniqueness of the solution, completing the decoding process.

Cyclic Codes

Another important family of codes is that of cyclic codes and their quasi-cyclic generalizations. Cyclic codes are a widely used family of error-correcting codes whose codewords remain valid under cyclic shifts. This simple structural property makes them efficient to implement and analyze.
Let $\mathbf{c} = (c_0, \ldots, c_{n-1}) \in \mathbb{F}_q^n$, then we denote by $\sigma(c)$ its cyclic shift, i.e.,

$$\sigma(c_0, \ldots, c_{n-1}) = (c_{n-1}, c_0, \ldots, c_{n-2}).$$

A code is called cyclic, if the cyclic shift of any codeword is still a codeword.

Definition 19 (Cyclic Code). Let $\mathscr{C}$ be an $[n,k]$-linear code over $\mathbb{F}_q$. We say that $\mathscr{C}$ is cyclic if $\sigma(\mathscr{C}) = \mathscr{C}$.

A vector $\mathbf{c}$ can be associated with a polynomial in $\mathbb{F}_q[x]$ via the natural identification

$$c(x) = c_0 + c_1 x + c_2 x^2 + \cdots + c_{n-1} x^{n-1}.$$

Indeed, this map is an isomorphism between $\mathbb{F}_q^n$ and the ring of polynomials with coefficients in $\mathbb{F}_q$ and degree less than n.

Under this correspondence, the cyclic shift of $\mathbf{c}$ corresponds to multiplication by x modulo $x^n - 1$, since

$$x \cdot c(x) = c_{n-1} + c_0 x + \cdots + c_{n-2} x^{n-1} \pmod{x^n - 1}.$$

Consequently, a cyclic code $\mathscr{C}$ of length n over $\mathbb{F}_q$ corresponds to an ideal in the quotient ring $\mathbb{F}_q[x]/(x^n - 1)$. As $\mathbb{F}_q[x]$ is a principal ideal domain, every such ideal is generated by a single polynomial.

Definition 20 (Generator Polynomial). A generator polynomial of a cyclic code $\mathscr{C} \subset \mathbb{F}_q^n$ is a generator of minimal degree of the corresponding ideal in $\frac{\mathbb{F}_q[x]}{(x^n-1)}$.

Let $\mathscr{C}$ be a cyclic code over $\mathbb{F}_q$ of length n with generator polynomial $g(x) = \sum_{i=0}^{r} g_i x^i$, where r is the degree of g, the following fundamental properties hold.:

1. $g(x)$ divides $x^n - 1$ in $\mathbb{F}_q[x]$;
2. The dimension of $\mathscr{C}$ is $k = n - r$;
3. A generator matrix $G \in \mathbb{F}_q^{(n-r)\times n}$ of $\mathscr{C}$ is given by

$$G = \begin{pmatrix} g_0 & \cdots & g_r & & \\ & \ddots & & \ddots & \\ & & g_0 & \cdots & g_r \end{pmatrix}.$$

4. Let $h(x) \in \mathbb{F}_q[x]$ be such that $g(x)h(x) = x^n - 1$. Then the dual code $\mathscr{C}^\perp$ is also cyclic and is generated by the reciprocal polynomial of $h(x)$. In particular, $h(x)$ is called the parity-check polynomial of $\mathscr{C}$.

The cyclic structure allows a compact description using rotation matrices.

Definition 21 (Rotation Matrix). Let $\mathbf{u}, \mathbf{v} \in \mathbb{F}_q^n$ and define the rotation matrix as

$$\mathrm{rot}(\mathbf{u}) = \begin{pmatrix} \mathbf{u} \\ \sigma(\mathbf{u}) \\ \vdots \\ \sigma^{n-1}(\mathbf{u}) \end{pmatrix}.$$

The product $\mathbf{u}\mathrm{rot}(\mathbf{v})$ corresponds to the cyclic convolution of $\mathbf{u}$ and $\mathbf{v}$, which is equivalent to the multiplication of the associated polynomials modulo $x^n - 1$.

Cyclic codes can be generalized by relaxing the shift invariance condition, defining the quasi-cycling codes.

Definition 22 (ℓ-Cyclic Shift). For $\mathbf{x} = (x_1, \ldots, x_n) \in \mathbb{F}_q^n$ and an integer $\ell \geq 1$, the ℓ-cyclic shift is

$$\sigma_\ell(\mathbf{x}) = (x_{1+\ell}, \ldots, x_{n+\ell}),$$

where indices are taken modulo n.

Definition 23 (Quasi-Cyclic Code). An $[n,k]$ linear code $\mathscr{C}$ over $\mathbb{F}q$ is quasi-cyclic (QC) of index ℓ if $\sigma_\ell(\mathscr{C}) = \mathscr{C}$.

If $\mathscr{C}$ is a QC code of length $n = \ell a$ and index ℓ, then its generator matrix can be written in block form as

$$G = [C_1|C_2|\ldots|C_a],$$

where each C_i is an $a \times a$ circulant matrix. A circulant matrix is fully determined by its first row; each subsequent row is a cyclic shift of the previous one. This block-circulant structure enables compact storage and efficient computation, as each circulant block requires only a field elements for its description.

1.3 Cryptography

In order to introduce the cryptosystems based on coding theory, a brief introduction to cryptography is necessary. Cryptography is the mathematical discipline that studies techniques for performing secure transformations through encryption and decryption algorithms. It enables two parties to exchange information privately and verify its integrity, even if an eavesdropper can observe their communication.

A fundamental distinction is made between two types of encryption:

- **Symmetric cryptography**, also known as private-key cryptography, uses the same secret key for both encryption and decryption, or, more generally, uses two keys that are computationally easy to derive from one another. While efficient, this approach requires the two communicating parties to share a secret key in advance, which must be exchanged through a secure channel. This is a significant practical challenge in many real-world scenarios;
- **Asymmetric cryptography**, also called public-key cryptography, uses two keys: a public key (used to encrypt) and a private key (used to decrypt). The public key can be shared, eliminating the need for a secure initial key exchange. Moreover, it is computationally infeasible to derive the private key from the public key, as doing so would require solving a NP problem.

1.3.1 Public-Key Encryption (PKE)

Asymmetric encryption, or public-key encryption (PKE), typically involves two legitimate parties who wish to exchange messages, Alice (the receiver) and Bob (the sender), and a malicious third party, Eve, who attempts to eavesdrop on the communication.
The algorithm begins with a key-generation phase performed by the receiver. Alice

generates a secret (private) key sk and a corresponding public key pk. The two keys form a matched pair; only the private key can decrypt messages encrypted with the associated public key. The public key is made available to anyone, so Bob can use it to encrypt a message m into a ciphertext c and send c to Alice. Upon receiving the ciphertext, Alice uses her private key to recover the original message.
During the transmission, an attacker Eve may obtain the ciphertext c and the public key pk. The encryption scheme is considered secure if, given only c and pk, Eve cannot feasibly derive the private key sk or the plaintext m.

Table 1.1 shows a scheme of the messages exchanged and the operations performed during PKE.

ALICE		BOB
KEY GENERATION		
Generate key pair $(\mathsf{pk}, \mathsf{sk})$		
	$\xrightarrow{\mathsf{pk}}$	
		ENCRYPTION
		Choose a message m
		Compute $c = \mathsf{Enc}_{\mathsf{pk}}(m)$
	$\xleftarrow{c}$	
DECRYPTION		
Compute $m = \mathsf{Dec}_{\mathsf{sk}}(c)$		

Table 1.1 Public-Key Encryption

More formally, a public-key encryption scheme is defined as a triple of probabilistic polynomial-time (PPT) algorithms $(\mathsf{KeyGen}, \mathsf{Enc}, \mathsf{Dec})$.

Definition 24 (Public-Key Encryption Scheme (PKE)). A public-key encryption scheme Π is a triple of algorithms:

- $\mathsf{KeyGen}(1^\lambda) \to (\mathsf{pk}, \mathsf{sk})$: the key generation algorithm takes a security parameter λ (in unary) and outputs a public key pk and a secret key sk;
- $\mathsf{Enc}_{\mathsf{pk}}(m) \to c$: the encryption algorithm takes a public key pk and a message m from the message space $\mathscr{M}$, and outputs a ciphertext c;

- $\mathsf{Dec}_{\mathsf{sk}}(c) \to m/\perp$: the decryption algorithm takes a secret key sk and a ciphertext c, and outputs either a message $m \in \mathscr{M}$ or a special symbol $\perp$ indicating failure.

The scheme must satisfy the correctness property: For any $(\mathsf{pk}, \mathsf{sk})$ generated by $\mathsf{KeyGen}(1^\lambda)$ and any message $m \in \mathscr{M}$,

$$\Pr\left[\mathsf{Dec}_{\mathsf{sk}}(\mathsf{Enc}_{\mathsf{pk}}(m)) = m\right] = 1.$$

One-Way Functions and Trapdoors

The security of most post-quantum public-key cryptosystems relies on the concept of one-way functions and trapdoors.
A one-way function is a mathematical function that is easy to calculate but computationally difficult to invert: given $y = f(x)$, finding any preimage x' such that $f(x') = y$ is infeasible without additional information.
On the other hand, a trapdoor is a secret information, usually the private key, that makes inversion easy for the legitimate user.
In a PKE scenario, a one-way function that relies on an NP problem is applied in the Enc algorithm while the trapdoor is exploited in the Dec algorithm by the legitimate owner of the secret key.
The security reduction usually shows that breaking the scheme is as hard as solving the underlying NP problem without the trapdoor. This framework ensures that even in a post-quantum era, where Shor's algorithm breaks number-theoretic schemes, code-based systems remain secure if the underlying one-way function resists quantum attacks.

1.3.2 Key Encapsulation Mechanism (KEM)

While public-key encryption schemes allow the direct encryption of arbitrary messages, in practice they are often inefficient for large payloads. For this reason, modern cryptographic protocols separate the problem of establishing a shared secret from that of encrypting data. This separation is achieved through a Key Encapsulation Mechanism (KEM).

A KEM is a public-key primitive whose purpose is not to encrypt a message, but to securely establish a shared symmetric secret key between two parties over an insecure channel. This shared key can then be used within a symmetric encryption scheme, which is typically much more efficient. As a result, KEMs are a fundamental building block in hybrid encryption systems and are widely used in secure communication protocols.

As in public-key encryption, Alice first runs a key generation algorithm to produce a public/secret key pair $(\mathsf{pk}, \mathsf{sk})$ and publishes pk. To establish a shared secret, Bob runs an encapsulation algorithm using Alice's public key. This algorithm outputs a ciphertext c and a session key K. Bob sends the ciphertext c to Alice, who then applies a decapsulation algorithm using her secret key sk to recover the same session key K. An adversary observing the public key and the ciphertext should not be able to obtain any information about K.

Table 1.2 shows a scheme of the messages exchanged and the operations performed during KEM and the subsequent symmetric encryption and decryption.

More formally, a key encapsulation mechanism is defined as a triple of probabilistic polynomial-time algorithms $(\mathsf{KeyGen}, \mathsf{Encaps}, \mathsf{Decaps})$.

Definition 25 (Key Encapsulation Mechanism (KEM)). A key encapsulation mechanism $\mathscr{K}$ consists of the following algorithms:

- $\mathsf{KeyGen}(1^\lambda) \to (\mathsf{pk}, \mathsf{sk})$: the key generation algorithm takes a security parameter λ (in unary) and outputs a public key pk and a secret key sk;
- $\mathsf{Encaps}(\mathsf{pk}) \to (c, \mathsf{K})$: the encapsulation algorithm takes the public key pk and outputs a ciphertext c together with a session key K;
- $\mathsf{Decaps}(\mathsf{sk}, c) \to \mathsf{K}/\perp$: the decapsulation algorithm takes the secret key sk and a ciphertext c, and outputs the corresponding session key K or a failure symbol $\perp$.

The correctness requirement states that for any key pair $(\mathsf{pk}, \mathsf{sk})$ generated by $\mathsf{KeyGen}(1^\lambda)$,

$$\Pr_{(c,\mathsf{K}) \leftarrow \mathsf{Encaps}(\mathsf{pk})} [\mathsf{Decaps}(\mathsf{sk}, c) = \mathsf{K}] = 1.$$

The encapsulation algorithm is probabilistic and internally samples randomness. Equivalently, one may view $(c, \mathsf{K}) \leftarrow \mathsf{Encaps}(\mathsf{pk}, r)$ for uniformly random value r.

ALICE		BOB
KEY GENERATION		
Generate key pair $(\mathsf{pk}, \mathsf{sk})$		
	$\xrightarrow{\mathsf{pk}}$	
		ENCAPSULATION
		Compute $(c, \mathsf{K}) = \mathsf{Encaps}(\mathsf{pk})$
	$\xleftarrow{c}$	
DECAPSULATION		
Compute $\mathsf{K} = \mathsf{Decaps}(\mathsf{sk}, c)$		
		SYMMETRIC ENCRYPTION
		Choose a message m
		Compute $c_s = \mathsf{Enc}_{\mathsf{K}}(m)$
	$\xleftarrow{c_s}$	
SYMMETRIC DECRYPTION		
Compute $m = \mathsf{Dec}_{\mathsf{K}}(c_s)$		

Table 1.2 Key Encapsulation Mechanism with Symmetric Encryption

The exact manner in which c and K are derived from pk depends on the different KEM construction.

In post-quantum cryptography, KEMs have emerged as the preferred application for public-key primitives. Compared to traditional PKE, KEMs offer a simpler security interface and combine naturally with symmetric encryption schemes forming secure and efficient hybrid protocols.

1.4 Security Considerations

1.4.1 Complexity Classes

Computational complexity theory studies the difficulty of solving computational problems in terms of the resources required by algorithms, such as time and space.

By classifying problems based on their complexity class, it is possible to understand which problems are practically solvable and which are intractable for large inputs.

The standard model underlying complexity theory is the Turing machine, an abstract mathematical model of computation introduced by Alan Turing in 1936. A Turing machine consists of:

- An infinite tape divided into cells;
- A read/write head that can move left or right along the tape;
- A finite set of states, including a special start state and accept/reject states.

According to the Church-Turing thesis, any function that can be computed by any reasonable model of computation can be computed by a Turing machine.

The complexity classes are mainly divided into two families: P and NP class.

Definition 26 (P Class). P denotes the class of problems that can be solved by a deterministic Turing machine in polynomial time.

Formally, a problem is in P if there exists an algorithm that solves it in $\mathscr{O}(n^k)$ time for some constant k, where n is the input size.
This types of problems are considered tractable, as their running time grows at a manageable rate as input size increases.

Definition 27 (NP Class). NP denotes the class of problems that can be solved by a non-deterministic Turing machine in polynomial time.

In other words, NP contains problems whose solutions can be quickly checked once found, even if finding those solutions might be difficult.

It is possible to strengthen the definition of the NP class by introducing the NP-hard class.

Definition 28 (NP-hard Class). A problem $\mathscr{P}$ is NP-hard if for every problem $\mathscr{R}$ in NP, there exists a polynomial-time reduction from $\mathscr{R}$ to $\mathscr{P}$.

Polynomial-time reduction from $\mathscr{R}$ to $\mathscr{P}$ means that if one can solve $\mathscr{P}$, one can also solve $\mathscr{R}$ and thus solving $\mathscr{P}$ is at least as hard as solving $\mathscr{R}$.

NP-hard problems are at least as difficult as the hardest problems in NP. Solving any NP-hard problem efficiently would imply that all NP problems can be solved efficiently.
It is important to notice that NP-hard problems do not necessarily have to be in NP themselves. As a consequence, the NP-complete class has been introduced, which denotes the intersection of NP-hard and NP classes.

Definition 29 (NP-complete Class). A problem $\mathscr{P}$ is NP-complete, if it is NP-hard and in NP.

NP-complete problems represent the most difficult problems within NP—if any NP-complete problem could be solved in polynomial time, then P would equal NP.

1.4.2 Computational Problems in Coding Theory

Nearest Codeword Problem (NCP) and Syndrome Decoding Problem (SDP)

The security of code-based cryptography is based on the conjectured computational intractability of certain problems for random linear codes. The most fundamental is the nearest codeword problem, which consists of finding the codeword closest to a given received word in terms of Hamming distance.

Definition 30 (Nearest Codeword Problem (NCP)). Given a generator matrix $G \in \mathbb{F}_q^{k\times n}$ of a linear code $\mathscr{C}$ and a vector $\mathbf{y} \in \mathbb{F}_q^n$, the nearest decoding problem asks to find a codeword $\mathbf{c} \in \mathscr{C}$ that minimizes the Hamming distance $d(\mathbf{y},\mathbf{c})$.

This is equivalent to finding an error vector $\mathbf{e} \in \mathbb{F}_q^n$ of minimum Hamming weight such that $\mathbf{y}-\mathbf{e} \in \mathscr{C}$.

A related formulation using the parity-check matrix is often more convenient for cryptographic applications. To introduce it, it is useful to recall the concept of a syndrome.
Let $\mathscr{C}$ be a linear $[n,k]$-code with parity-check matrix H. The syndrome is the linear map

$$s : \mathbb{F}_q^n \to \mathbb{F}_q^{n-k}, \quad s(\mathbf{y}) = \mathbf{y}H^\top.$$

For a received word $\mathbf{y} = \mathbf{c}+\mathbf{e}$, with $\mathbf{c} \in \mathscr{C}$ and error vector $\mathbf{e}$, one has

$$s(\mathbf{y}) = s(\mathbf{e}),$$

since $\mathbf{c}H^\top = 0$. Hence, the syndrome depends only on the error pattern and not on the transmitted codeword.
This property is the foundation of syndrome-based decoding: errors can be identified and corrected by matching the computed syndrome to a known error pattern.

Definition 31 (Syndrome Decoding Problem (SDP)). Given a parity-check matrix $H \in \mathbb{F}_q^{(n-k)\times n}$ of a linear code $\mathscr{C}$, a syndrome vector $\mathbf{s} \in \mathbb{F}_q^{n-k}$, and an integer $t > 0$, the syndrome decoding problem asks to find an error vector $\mathbf{e} \in \mathbb{F}_q^n$ with Hamming weight $\mathrm{wt}_H(\mathbf{e}) \leq t$ such that $H\mathbf{e}^\top = \mathbf{s}^\top$.

In essence, SDP reformulates NCP in syndrome language: instead of searching directly for the nearest codeword, one searches for a low-weight error pattern that explains the observed syndrome. The two problems are computationally equivalent under the bounded-error model.

In practice, syndrome decoding proceeds as follows:

1. Pre-compute the syndromes of all error patterns that the code can correct;
2. For a received word $\mathbf{y}$, compute its syndrome $s(\mathbf{y})$;
   (a) If $s(\mathbf{y}) = \mathbf{0}$, then $\mathbf{y}$ is a codeword and no error has occurred;
   (b) Otherwise, check whether $s(\mathbf{y})$ matches one of the pre-computed syndromes. If a match is found, the corresponding error pattern can be subtracted from $\mathbf{y}$ to recover the original codeword.

While this approach works well for small codes, it becomes impractical for cryptographic parameters due to the exponential size of the syndrome table. This computational difficulty is precisely what provides the security foundation for code-based cryptography.

The decision versions of both the Nearest Codeword Problem and the Syndrome Decoding Problem are known to be NP-complete for binary linear codes. This fundamental result was first proven by Berlekamp, McEliece, and van Tilborg in 1978 [3].

Linear Code Equivalence Problem (LCEP)

Another fundamental problem is the linear code equivalence problem.
In order to define the problem, some preliminary notions need to be introduced.

Definition 32 (Isometry). Let V be a vector space over $\mathbb{F}_q$ endowed with a distance function d. A map $\varphi : V \to V$ is called an isometry if it preserves the distance, that is,

$$d(x,y) = d(\varphi(x), \varphi(y)) \quad \text{for all } x, y \in V.$$

The set of all isometries with respect to the distance d is denoted by $\mathscr{I}_d$.

For the vector space $V = \mathbb{F}_q^n$ equipped with the Hamming metric, all linear maps that preserve Hamming distance consist of monomial transformations together with field automorphisms applied to each entry.

In cryptography, the focus is on the monomial transformations, which are a subclass of these isometries defined by:

$$M_{n,q} = (\mathbb{F}_q^*)^n \rtimes S_n,$$

where

- $(\mathbb{F}_q^*)^n$ is the group of n-tuples of non-zero scalars;
- S_n is the group of permutations of $\{1, \ldots, n\}$;
- $\rtimes$ is the semidirect product.

In conclusion, $M_{n,q}$ is exactly the set of all linear maps that preserve Hamming weight for every vector.

Any monomial transformation $\varphi \in M_{n,q}$ can be written as multiplication by an invertible matrix $M = PD$, where:

- $D = \mathrm{diag}(v)$ is an $n \times n$ diagonal matrix whose diagonal entries come from a vector $v \in (\mathbb{F}_q^*)^n$. This matrix scales each coordinate of a vector independently by a non-zero element of $\mathbb{F}_q$;
- P is an $n \times n$ permutation matrix (each row and each column contains exactly one 1 and zeros elsewhere). Multiplication by P permutes the coordinates of the vector.

Definition 33 (Code Equivalence). Let $\mathscr{C}_1, \mathscr{C}_2 \subseteq \mathbb{F}_q^n$ be linear codes. $\mathscr{C}_1$ and $\mathscr{C}_2$ are said equivalent if there exists an isometry $\varphi \in \mathscr{I}_d$ such that

$$\varphi(\mathscr{C}_1) = \mathscr{C}_2.$$

In the Hamming metric, this notion specializes to equivalence under monomial transformations.
Since monomial transformations consist of permutations and scalings, two common subclasses of equivalence are distinguished.

Definition 34 (Permutation Equivalence). Two codes $\mathscr{C}_1, \mathscr{C}_2 \subseteq \mathbb{F}_q^n$ are permutation equivalent if there exists a permutation $\sigma \in S_n$ such that

$$\sigma(\mathscr{C}_1) = \mathscr{C}_2.$$

Definition 35 (Linear (Monomial) Equivalence). Two codes $\mathscr{C}_1, \mathscr{C}_2 \subseteq \mathbb{F}_q^n$ are linearly equivalent if there exists a monomial transformation

$$\varphi \in (\mathbb{F}_q^*)^n \rtimes S_n$$

such that

$$\varphi(\mathscr{C}_1) = \mathscr{C}_2.$$

Clearly, permutation equivalence is a special case of linear equivalence.

Let $C \subseteq \mathbb{F}_q^n$ be a linear $[n,k]$-code with generator matrix $G \in \mathbb{F}_q^{k\times n}$.
Left-multiplication by an invertible matrix $M \in \mathbb{F}_q^{k\times k}$ corresponds to choosing a different basis of the same code, while right-multiplication by a permutation matrix $P \in \mathbb{F}_q^{n\times n}$ corresponds to a permutation of coordinates, i.e., a code equivalence.
Thus, if

$$\hat{G} = MGP,$$

then G and $\hat{G}$ generate permutation-equivalent (and hence linearly equivalent) codes.

Definition 36 (Linear Code Equivalence Problem (LCEP)). Given a generator matrix $G \in \mathbb{F}_q^{k\times n}$, let $M \in \mathbb{F}_q^{k\times k}$ be an invertible matrix and let $P \in \mathbb{F}_q^{n\times n}$ be a permutation matrix. The linear code equivalence problem states that given the derived generator matrix $\hat{G} = MGP$ it is hard to derive G.

Regarding the complexity of the problem, it was proven that it cannot be easier than the graph isomorphism problem over $\mathbb{F}_q$, as there exist a polynomial time reduction from the linear code equivalence problem to the graph isomorphism problem over any field $\mathbb{F}_q$. The latter problem has been extensively studied, but until now there is no polynomial-time algorithm for solving all of its instances [4].

1.4.3 Formal Security Models

Cryptographic security is defined through precise mathematical models that formalize both adversarial capabilities and security goals. This section introduces the concept of indistinguishability and two of the main attack classes that fall into it, in order to detail three fundamentals formal security models. The presentation of the formal security models in this section is based on the lecture slides of Prof. Sanna [5].

Indistinguishability

One core concept underlying modern security definitions is indistinguishability, meaning whether an adversary can tell apart two different random variables. Concretely, in a secure encryption context, this translates to the requirement that an adversary eavesdropping on a communication cannot distinguish a genuine ciphertext from a string of random bits. Therefore, indistinguishability represents a foundational property for confidentiality, ensuring that the ciphertext reveals no meaningful information about the underlying plaintext.
This notion is first introduced into its most complete form, as perfect indistinguishability.

Definition 37 (Perfect or Information-Theoretical Indistinguishability). Let X an Y be two random variables. Then X and Y are perfectly indistinguishable if X and Y have identical probability distribution, that is

$$\Pr[X = x] = \Pr[Y = x],$$

for every x among over the possible values of X.

Informally, X and Y are indistinguishable if no adversary with unlimited resources can distinguish between samples of X and samples of Y.

Its clear that this level of security is ideal and too strict, as there not exist a computationally unbounded attacker in practical cryptosystems. For this reason, a more relaxed definition is presented.

Definition 38 (Computational Indistinguashibility). Let X and Y be discrete random variables depending on the security parameter λ. Then X and Y are computationally indistinguishable if for every Probabilistic Polynomial Time (PPT) algorithm $\mathscr{A}$ there exists a negligible function negl such that

$$|Pr[\mathscr{A}(X) = 1] - Pr[\mathscr{A}(Y) = 1]| \leq \mathsf{negl}(\lambda).$$

Informally, X and Y are computationally indistinguishable if no efficient adversary can distinguish between their samples with non-negligible advantage.
This relaxation from perfect to computational security enables the construction of practical cryptosystems.

Chose-Plaintext Attack (CPA)

A wide know and common family of attacks used against public-key cryptosystem is the chosen-plaintext attack, where the attacker take advantage of the public encryption in order to obtain the ciphertexts by means of arbitrary plaintexts. By exploiting this opportunity, the adversary can potentially compromise the security of the system. For this reason, security against this class of attacks is fundamental, and PKE schemes must be designed to be protected against any possible chosen-plaintext attack.
In this context, the notion of IND-CPA security emerged, which define if a PKE scheme has indistinguishable encryption under chosen-plaintext attacks.

A CPA experiment can be defined in terms of an indistinguishability game.

Definition 39 (CPA Indistinguishability Experiment). Let $\Pi = (\mathsf{KeyGen}, \mathsf{Enc}, \mathsf{Dec})$ be a public-key encryption scheme. For any adversary $\mathscr{A}$, the CPA indistinguishability experiment $\mathsf{PubK}^{cpa}_{\mathscr{A},\Pi}$ can be defined as follows:

1. A key pair $(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{KeyGen}(1^{\lambda})$ is generated;
2. The adversary $\mathscr{A}$ is given the public input 1^{λ} and the public key pk;

3. The adversary $\mathscr{A}$ outputs two messages m_0 and m_1 of equal length;
4. A random bit $b \xleftarrow{\$} 0,1$ is chosen, and a ciphertext $c \leftarrow \mathsf{Enc}_{\mathsf{pk}}(m_b)$ is computed and provided to $\mathscr{A}$;
5. The adversary $\mathscr{A}$ outputs a bit b' as its guess;
6. The experiment outputs 1 (success) if $b' = b$, and 0 (failure) otherwise.

It is now possible to detail a formal definition Of IND-CPA security,

Definition 40 (IND-CPA Security). A public-key encryption scheme Π is IND-CPA secure, if for every PPT adversary $\mathscr{A}$ there exists a negligible function negl such that

$$\Pr[\mathsf{PubK}^{cpa}_{\mathscr{A},\Pi}(\lambda) = 1] \leq \frac{1}{2} + \mathsf{negl}(\lambda).$$

Informally, an encryption scheme is IND-CPA secure if no probabilistic polynomial-time adversary can distinguish which of two chosen plaintexts corresponds to a given ciphertext, even when the adversary is allowed to choose the plaintexts itself.

Chosen-Ciphertext Attack (CCA)

Chosen-ciphertext attack (CCA) class is another fundamental attack typology, where an adversary $\mathscr{A}$ is able to obtain the decryption of arbitrary ciphertexts c' in order to recover fully or partially the target ciphertext c. It is important to underline that the adversary cannot query directly the original ciphertext, meaning that $c' \not\equiv c$.
Similarly to the CPA case, this attack model raised the necessity to design PKE scheme resistant to the threat, defining a new formal security level named IND-CCA security, under which PKE schemes have indistinguishable encryption under chosen-ciphertext attacks.

A new experiment must be constructed, always in terms of an indistinguishability game.

Definition 41 (CCA Indistinguishability Experiment). Let $\Pi = (\mathsf{KeyGen}, \mathsf{Enc}, \mathsf{Dec})$ be a public-key encryption scheme. For every adversary $\mathscr{A}$, the CCA indistinguishability experiment $\mathsf{PubK}^{cca}_{A,\Pi}$ can be defined as follows:

1. A key pair is generated $(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{KeyGen}(1^{\lambda})$;

2. The adversary $\mathscr{A}$ is given the public inputs 1^λ, the public key pk and access to a decryption oracle $\mathsf{Dec}_{\mathsf{sk}}$;
3. The adversary $\mathscr{A}$ outputs two messages m_0 and m_1 of equal length;
4. A random bit $b \xleftarrow{\$} \{0,1\}$ is chosen, and a ciphertext $c \leftarrow \mathsf{Enc}_{\mathsf{pk}}(m_b)$ is computed and provided to $\mathscr{A}$;
5. The adversary $\mathscr{A}$ repeatedly queries the oracle $\mathsf{Dec}_{\mathsf{sk}}$, without requesting the decryption of the target ciphertext c. Finally, $\mathscr{A}$ outputs a bit b' as its guess;
6. The experiment outputs 1 (success) if $b' = b$, and 0 (failure) if otherwise.

In this context, an oracle is an abstract entity that, given as input an instance of a specific computational problem, returns a corresponding solution.

The formal definition of IND-CCA security can be introduced by means of the CCA indistinguishability experiment.

Definition 42 (IND-CCA Security). A public-key encryption scheme Π is IND-CCA secure if for every PPT adversary $\mathscr{A}$ there exists a negligible function negl such that

$$\Pr\left[\mathsf{PubK}^{cca}_{\mathscr{A},\Pi}(\lambda) = 1\right] \leq \frac{1}{2} + \mathsf{negl}(\lambda).$$

Informally, an encryption scheme is CCA-secure if no probabilistic polynomial-time adversary can distinguish which of two chosen plaintexts corresponds to a given challenge ciphertext, even when the adversary is allowed to choose the plaintexts itself and has access to a decryption oracle for any ciphertext $c' \neq c$.

IND-CCA security was further divided into two branches:

- **IND-CCA1** corresponds to CCA-security with the restriction that the adversary is allowed to query the decryption oracle only before receiving the challenge ciphertext c;
- **IND-CCA2** corresponds to full chosen-ciphertext security and is also referred to as adaptive chosen-ciphertext attack security.

These security models form a strict hierarchy:

$$\text{IND-CCA2} \implies \text{IND-CCA1} \implies \text{IND-CPA}$$

with the reverse implications being false in general.
For practical deployment, IND-CCA2 security is considered the gold standard as it ensures security even when adversaries can obtain decryptions of related ciphertexts.

Formal Security Models for KEM Algorithms

The security notions introduced above for public-key encryption naturally extend to the key encapsulation setting, with appropriate adaptations to the security goal and adversarial interface. In a KEM scenario, the adversary's objective is no longer to distinguish between encryptions of chosen plaintexts, but rather to distinguish the encapsulated session key from a uniformly random key of the same length.
Accordingly, IND-CPA and IND-CCA security for KEMs are defined via indistinguishability games in which the challenge consists of a ciphertext–key pair (c, K) and the adversary must decide whether K is the genuine output of the encapsulation algorithm or a random key. Oracle access is adapted in the same manner: in the IND-CPA case, the adversary may generate encapsulations using the public key, while in the IND-CCA case it is additionally granted access to a decapsulation oracle, with the restriction that the challenge ciphertext cannot be queried. Despite these differences, the underlying structure of the security models remains unchanged, and the hierarchy between IND-CPA, IND-CCA1, and IND-CCA2 translates from public-key encryption to key encapsulation mechanisms.

1.4.4 Fujisaki-Okamoto Transformation

The Fujisaki-Okamoto (FO) transformation, originally introduced by Fujisaki and Okamoto in 1999 [6], represents a fundamental result in asymmetric cryptography, providing a generic method to convert a weakly secure encryption scheme into one that is secure against adaptive chosen-ciphertext attacks (IND-CCA) in the random oracle model. This result is particularly relevant in the context of post-quantum cryptography, where many schemes based on diverse mathematical problems offer security only against chosen-plaintext attacks (CPA).

Preliminary Definitions

To understand the FO transformation, several security notions and definitions are required.

Definition 43 (One-Way Encryption (OWE)). Let $\Pi = (\mathsf{KeyGen}, \mathsf{Enc}, \mathsf{Dec})$ be an asymmetric encryption scheme with message space MSPC, and let $\mathscr{A}$ be probabilistic polynomial-time adversary. For $k \in \mathbb{N}$, define the advantage of $\mathscr{A}$ by

$$\mathsf{Adv}^{OWE}_{\mathscr{A},\Pi}(k) = \Pr\Big[(\mathsf{pk},\mathsf{sk}) \leftarrow \mathsf{KeyGen}(1^k);\ x \leftarrow \mathsf{MSPC};\\ y \leftarrow \mathsf{Enc}_{\mathsf{pk}}(x) : \mathscr{A}(\mathsf{pk}, y) = x\Big].$$

It can be said that $\mathscr{A}$ $(t;\varepsilon)$-breaks Π if it runs in time at most t and achieves $\mathsf{Adv}^{OWE}_{\mathscr{A},\Pi}(k) \geq \varepsilon$. The scheme Π is (t,ε)-secure in the sense of OWE if no adversary can (t,ε)-break Π in that sense.

Informally, even given the public key and a ciphertext, a computationally bounded adversary cannot recover the original plaintext with probability significantly better than random guessing.

Definition 44 (Find-Guess Security (FG)). Let $\Pi = (\mathsf{Enc}, \mathsf{Dec})$ be a symmetric encryption scheme with key space KSPC and message space MSPC, and let $\mathscr{A}$ be a probabilistic polynomial-time adversary. For $k \in \mathbb{N}$, define the advantage of $\mathscr{A}$ by

$$\mathsf{Adv}^{FG}_{\mathscr{A},\Pi}(k) = 2 \cdot \Pr\Big[a \xleftarrow{\$} \mathsf{KSPC};\ (m_0, m_1, s) \leftarrow \mathscr{A}(\mathrm{find});\\ b \xleftarrow{\$} \{0,1\};\ y = \mathsf{Enc}_a(m_b) : \mathscr{A}(\mathrm{guess}, s, y) = b\Big] - 1.$$

It is required that for (m_0, m_1) that $\mathscr{A}$ outputs, $m_0 \neq m_1$ and $m_0, m_1 \in \mathsf{MSCP}$.
It can be said that $\mathscr{A}$ (t,ε)-breaks Π if it runs in time at most t and achieves $\mathsf{Adv}^{FG}_{\mathscr{A},\Pi}(k) \geq \varepsilon$. The scheme Π is (t,ε)-secure in the sense of FG if no adversary can (t,ε)-break Π in that sense.

Informally, even if an adversary can choose two messages and obtain an encryption of one of them, it cannot guess which one was encrypted with probability significantly better than $1/2$.

Definition 45 (γ-Uniformity). Let $\Pi = (\mathsf{KeyGen}, \mathsf{Enc}, \mathsf{Dec}, \mathsf{COINS}, \mathsf{MSPC})$ be an asymmetric encryption scheme. For given $(\mathsf{pk},\mathsf{sk}) \leftarrow \mathsf{KeyGen}(1^k)$, $x \in \mathsf{MSPC}$, and

$y \in \{0,1\}^*$ define

$$\gamma(x,y) = \Pr[h \xleftarrow{\$} \mathsf{COINS} : y = \mathsf{Enc}_{\mathsf{pk}}(x;h)].$$

It can be said that Π is γ-uniform (for $k \in \mathbb{N}$), if, for any $(\mathsf{pk},\mathsf{sk}) \leftarrow \mathsf{KeyGen}(1^k)$, any $x \in \mathsf{MSPC}$ and any $y \in \{0,1\}^*$, $\gamma(x,y) \leq \gamma$.

Here, COINS denotes the randomness space of the encryption algorithm, i.e., the set of all possible random strings used internally by Enc. The notation $\mathsf{Enc}_{\mathsf{pk}}(x;h)$ makes explicit that encryption is a randomized procedure depending on random coins $h \in \mathsf{COINS}$.

Informally, γ-uniformity bounds the probability that a fixed plaintext x encrypts to a specific ciphertext y over the random coins of the encryption algorithm. In other words, no ciphertext can occur with probability greater than γ when encrypting a given message. This property ensures that the randomness used in the encryption process spreads ciphertexts "uniformly enough", preventing high-probability outputs that could leak structural information.

The FO transformation utilizes two hash functions:

- $\mathscr{G} : \mathsf{MSPC}^{\mathsf{asym}} \to \mathsf{KSPC}^{\mathsf{sym}}$ maps asymmetric plaintexts to symmetric keys;
- $\mathscr{H} : \mathsf{MSPC}^{\mathsf{asym}} \times \mathsf{MSPC}^{\mathsf{sym}} \to \mathsf{COINS}^{\mathsf{asym}}$ maps message pairs to encryption randomness.

In the random oracle model, both $\mathscr{G}$ and $\mathscr{H}$ are modeled as ideal random functions.

Many encryption schemes, including most post-quantum candidates, are only proven secure against chosen-plaintext attacks (CPA). However, real-world protocols and standards require security against chosen-ciphertext attacks (CCA).
The FO transformation addresses this need by providing a procedure that takes as inputs:

- A weakly secure asymmetric encryption scheme (OWE-secure);
- A weakly secure symmetric encryption scheme (FG-secure),

and outputs a strongly secure hybrid encryption scheme that is IND-CCA secure in the random oracle model.

The Basic FO Construction

The FO transformation constructs a hybrid encryption scheme

$$\Pi^{\mathsf{hy}} = (\mathsf{KeyGen}^{\mathsf{hy}}, \mathsf{Enc}^{\mathsf{hy}}, \mathsf{Dec}^{\mathsf{hy}}, \mathsf{MSPC}^{\mathsf{hy}}, \mathsf{COINS}^{\mathsf{hy}})$$

from an asymmetric scheme and a symmetric scheme

$$\Pi^{\mathsf{asym}} = (\mathsf{KeyGen}^{\mathsf{asym}}, \mathsf{Enc}^{\mathsf{asym}}, \mathsf{Dec}^{\mathsf{asym}}, \mathsf{MSPC}^{\mathsf{asym}}, \mathsf{COINS}^{\mathsf{asym}});$$
$$\Pi^{\mathsf{sym}} = (\mathsf{Enc}^{\mathsf{sym}}, \mathsf{Dec}^{\mathsf{sym}}, \mathsf{MSPC}^{\mathsf{sym}}, \mathsf{KSPC}^{\mathsf{sym}}),$$

where $\mathsf{COIN}^{\mathsf{hy}} = \mathsf{MSPC}^{\mathsf{asym}}$ and $\mathsf{MSPC}^{\mathsf{hy}} = \mathsf{KSPC}^{\mathsf{sym}}$.
Let $m \in \mathsf{MSPC}^{hy}$, the transformation works as follows:

- **Key Generation:** $\mathsf{KeyGen}^{\mathsf{hy}}(1^{\lambda}) \to (\mathsf{pk}, \mathsf{sk})$
  1. Execute the asymmetric key generation: $(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{KeyGen}^{\mathsf{asym}}(1^{\lambda})$;
  2. Output the same key pair $(\mathsf{pk}, \mathsf{sk})$.
- **Encryption:** $\mathsf{Enc}^{\mathsf{hy}}_{\mathsf{pk}}(m) \to c$
  1. Choose a random seed $\sigma \xleftarrow{\$} \mathsf{MSPC}^{\mathsf{asym}}$;
  2. Derive encryption randomness $r \leftarrow \mathscr{H}(\sigma, m)$;
  3. Derive symmetric key $\mathsf{k} \leftarrow \mathscr{G}(\sigma)$;
  4. Compute asymmetric component $c_1 \leftarrow \mathsf{Enc}^{\mathsf{asym}}_{\mathsf{pk}}(\sigma;\, r)$;
  5. Compute symmetric component $c_2 \leftarrow \mathsf{Enc}^{\mathsf{sym}}_{\mathsf{k}}(m)$;
  6. Output ciphertext $c = (c_1, c_2)$.
- **Decryption:** $\mathsf{Dec}^{\mathsf{hy}}_{\mathsf{sk}}(c) \to m$ or $\perp$
  1. Decrypt the asymmetric part $\sigma' \leftarrow \mathsf{Dec}^{\mathsf{asym}}_{\mathsf{sk}}(c_1)$;
  2. If decryption fails, output $\perp$;
  3. Compute symmetric key $\mathsf{k}' \leftarrow \mathscr{G}(\sigma')$;
  4. Decrypt the symmetric part $m' \leftarrow \mathsf{Dec}^{\mathsf{sym}}_{\mathsf{k}'}(c_2)$;

5. Compute $r' \leftarrow \mathscr{H}(\sigma', m')$ and verify

$$\mathsf{Enc}_{\mathsf{pk}}^{\mathsf{asym}}(\sigma';\, r') = c_1;$$

6. If verification succeeds, output m'; otherwise output $\perp$.

The encryption process can be visualized as:

$$\mathsf{Enc}_{\mathsf{pk}}^{\mathsf{hy}}(m) = \mathsf{Enc}_{\mathsf{pk}}^{\mathsf{asym}}(\sigma;\, \mathscr{H}(\sigma, m)) \parallel \mathsf{Enc}_{\mathscr{G}(\sigma)}^{\mathsf{sym}}(m),$$

where σ acts as a random session key that is asymmetrically encrypted.

The security of FO transformation is stated in the following theorem, which establishes that if Π^{asym} is γ-uniform and OWE-secure, and Π^{sym} is FG-secure, then the resulting hybrid scheme is IND-CCA secure with concrete bounds depending on γ, the number of random oracle queries, and the number of decryption queries. Note that here the FO transformation's hash function $\mathscr{G}$ and $\mathscr{H}$ are treated as random oracles.

Theorem 3 (Chosen-Ciphertext Security). Suppose Π^{asym} is γ-uniform and (t_1, ε_1)-secure in the sense of OWE. Suppose Π^{sym} is (t_2, ε_2)-secure in the sense of FG. Let ℓ_1 and ℓ_2 be the sizes of $\mathsf{MSPC}^{\mathsf{asym}}$ and $\mathsf{MSPC}^{\mathsf{sym}}$, respectively.
Then Π^{hy} is $(t, q_g, q_h, q_d, \varepsilon)$-secure in the sense of IND-CCA in the random oracle model, where

$$\begin{aligned} t &= \min(t_1, t_2) - O\big((q_g + q_h)(\ell_1 + \ell_2)\big), \\ \varepsilon &= \Big(2(q_g + q_h)\varepsilon_1 + \varepsilon_2 + 1\Big)\Big(1 - 2\varepsilon_1 - 2\varepsilon_2 - \gamma - 2^{-\ell_2}\Big)^{q_d} - 1, \end{aligned}$$

and

- q_g is the number of queries to the random oracle $\mathscr{G}$;
- q_h is the number of queries to the random oracle $\mathscr{H}$;
- q_d is the number of decryption queries, meaning the CCA power.

In particular

- Π^{asym} is γ-uniform ensures that no ciphertext occurs with too large probability, preventing structural bias in asymmetric encryption outputs;

- (t_1, ε_1)-secure in OWE ensures that the asymmetric scheme is hard to invert, meaning that given pk and c, recovering the plaintext is hard;
- (t_2, ε_2)-secure in FG ensures that the symmetric scheme hides which message was encrypted, blocking any leak of information in the second ciphertext component.

Chapter 2

Code-Based Public-Key Encryption Frameworks

2.1 Post-Quantum Cryptographic Frameworks

In the last decades, public key cryptography has become an indispensable component of the global communication digital infrastructure. The majority of applications implemented in society, such as Internet, mobile phones, cloud computing and online commerce, rely on these cryptographic networks. In this scenario, security owns a central relevance, as those services loose their validity if not considered secure. For this reason cryptographic techniques like PKE, digital signatures and key exchange have been implemented in the digital infrastructure.
Before the advent of quantum computers and quantum algorithms, the classical cryptographic techniques were based on the conjecture that it is ”hard” to solve any of the following some mathematical problems in a reasonable time, such as factoring a large integer, order-finding or finding a discrete logarithm. The assumption is that these problems cannot be solved in reasonable complexity, i.e. polynomial time. In fact, all known classical computer algorithms require at least exponential time $\mathcal{O}(n2^n)$ to solve the previous hard problems. However, quantum computers only require a polynomial complexity of $\mathcal{O}(n^2)$, rendering all public key cryptosystems based on such assumptions vain.

The key difference between classical and quantum computing lies in their basic unit of information. While classical computers are based on binary bits, quantum

computers use qubits capable of superposition, meaning holding multiple potential states at once, to explore many computational paths in parallel. This property permits the quantum computers to perform calculations in a different and faster way, as demonstrated in the Shor's algorithm.

In 1994, Peter Shor [7] designed its revolutionary algorithm which aim is to efficiently factorize huge numbers, operation that was previously impossible for any classical supercomputer. The step-by-step procedure comprehend some classical operation and a fundamental quantum-based step. The core principle of the quantum operation is to determine the period of a specific function using the Quantum Fourier Transform (QFT). The QFT is used to analyze a function $f(x)$ and identify its period. The presentation of the algorithm below is inspired by [8]. Shor's algorithm works as follows:

1. **Select a co-prime integer.** Let n be the number that needs to be factorized. Choose a random integer m such that $1 < m < n$ and compute the greatest common divisor $\gcd(m,n)$ using the Euclidean algorithm, a efficient classical algorithm.

   - If $\gcd(m,n) \neq 1$, then m itself is a non-trivial factor of n, and the process is complete;
   - If $\gcd(m,n) = 1$ (meaning m and n are co-prime), proceed to the next step.

2. **Quantum period-finding.** This is the only quantum-based step, where a quantum computer is employed to find the period φ of the sequence generated by the function $f(x) = m^x \mod n$. This sequence is:

   $$m^1 \pmod n, \quad m^2 \pmod n, \quad m^3 \pmod n, \quad \ldots$$

   The value φ is the smallest positive integer for which $m^\varphi \mod n = 1$;

3. **Verify period parity.** Check if the found period φ is even.

   - If φ is odd, return to Step 1 and choose a new random integer m;
   - If φ is even, proceed. Given that φ is even, then $m^\varphi \equiv 1 \pmod n$, which implies $(m^{\varphi/2} - 1)(m^{\varphi/2} - 1) \equiv 0 \pmod n$. Therefore, n must divide this product.

4. **Check for triviality.** To avoid a trivial factor (n itself), check if $m^{\varphi/2}+1$ is a multiple of n.
   - If $m^{\varphi/2}+1 \equiv 0 \pmod{n}$ (i.e., n divides $m^{\varphi/2}+1$), the factor is trivial. Return to Step 1;
   - If $m^{\varphi/2}+1 \not\equiv 0 \pmod{n}$, proceed. This condition ensures that n does not divide $m^{\varphi/2}+1$, so it must share a factor with $m^{\varphi/2}-1$.
5. **Compute the factor.** Finally, calculate $d = \gcd(m^{\varphi/2}-1, n)$ using the Euclidean algorithm. Since it was established in Step 4 that n does not divide $m^{\varphi/2}+1$, the result d will be a non-trivial prime factor of n.

The quantum speedup comes from using the QFT in Step 2 to find the period φ exponentially faster than any known classical algorithm. Once φ is found, the rest of the steps are classical and efficient. This permits to solve classical hard problem like factorization in a reasonable time instead of an exponential one.

The advent of quantum algorithms that achieve exponential speedup determine a great threat to the classical standards based on the previous cited hard problems. Current physical quantum computers have yet to reach the hardware level required to efficiently implement the theoretical algorithms, however their future improvements will disrupt the current digital infrastructure.
Table 2.1 reports the impact of large-scale quantum computers on classical cryptographic algorithms.

Some countermeasures can be applied in this scenario. For standards as AES or the SHA family it is sufficient to increase the key sizes or the output size, respectively. Unfortunately, many other algorithm are no longer secure, so new standards needs to be created. Under this ideology, post-quantum cryptography was born. This new branch of research is the study of cryptographic algorithms (usually public-key) that are believed to be secure against both classical and quantum attacks. The new algorithms are based on quantum-hard mathematical fields, that are supposed to be resistant to the exponential speedup introduced by Shor.
Many families of post-quantum algorithms have been introduced, each one based on different hard problems:

- **Code-Based Cryptography**. This family was one the first proposed, and it relies on the hardness of decoding random-looking error-correcting codes. The

Cryptographic Algorithm	**Type**	**Purpose**	**Impact from large-scale quantum computer**
AES	Symmetric key	Encryption	Larger key sizes needed
SHA-2, SHA-3	Hash function	Hash functions	Larger output needed
RSA	Public key	Signatures, key establishment	No longer secure
ECDSA, ECDH (Elliptic Curve Cryptography)	Public key	Signatures, key exchange	No longer secure
DSA (Finite Field Cryptography)	Public key	Signatures, key exchange	No longer secure

Table 2.1 Impact of large-scale quantum computers on classical cryptographic algorithms [9]

associated hard problems are the syndrome decoding problem and the linear code equivalence problem;

- **Lattice-Based Cryptography**. This approach builds security on the apparent difficulty of solving certain complex problems in high-dimensional geometric lattices. It relies on different hard problems, such as the shortest vector problem and learning with errors;

- **Multivariate Polynomial Cryptography**. In a multivariate system, security comes from the difficulty of solving large systems of nonlinear polynomial equations. As a consequence, it relies on the multivariate quadratic polynomial problem and the isomorphism of polynomial problem;

- **Hash-Based Cryptography**. This family is used exclusively for digital signatures. Its security depends solely on the quantum-resistant properties of a underlying cryptographic hash function (like SHA-2 or SHA-3), so preimage resistance and collision resistance are its core problems;

- **Isogeny-Based Cryptography**. In this case elliptic curves are implemented, but in a different way than classic cryptography. Instead of relying on the discrete logarithm problem within a single curve, it uses the complex maps between curves, called isogenies. The hard problems are the supersingular

isogeny Diffie-Hellman problem and the computational supersingular isogeny problem.

In 2016, the National Institute of Standards and Technology (NIST) initiated its post-quantum cryptography standardization project, with the objective to evaluate and standardize one or more quantum-resistant public-key cryptographic algorithms for both Key Encapsulation Mechanisms and Digital Signatures. The selection process was structured as a multi-round public competition. Twenty-three signature schemes and fifty-nine encryption/KEM schemes were presented by the initial submission deadline. On August 13, 2024, NIST released final versions of the first three Post Quantum Crypto Standards after the third round end: FIPS 203, FIPS 204, and FIPS 205. Table 2.2 shows the details of the current selected algorithms. Other two standards have yet to be officialized; FIPS 206 for Falcon is currently in progress, while the code-based algorithm HQC has been recently selected during the ongoing fourth round.
On the other hand, Table 2.3 shows the remaining candidates of the fourth round: BIKE and McEliece have not been selected, but they are still under consideration for future standardization, while SIKE has been discarded as it was broken.

Algorithm Name	**Type**	**Category**	**Round**	**Standard (FIPS)**
CRYSTALS-Kyber	Lattice-based	PKE / KEM	Round 3	FIPS 203
CRYSTALS-Dilithium	Lattice-based	Signature	Round 3	FIPS 204
Falcon	Lattice-based	Signature	Round 3	FIPS 206
SPHINCS+	Hash-based	Signature	Round 3	FIPS 205
HQC	Code-based	PKE / KEM	Round 4	Not yet assigned

Table 2.2 NIST-standardized post-quantum cryptographic algorithms and their selection rounds

Algorithm	**Type**	**Category**	**Status After Round 4**
BIKE	Code-based	PKE/KEM	Candidate, not selected
Classic McEliece	Code-based	PKE/KEM	Candidate, not selected
SIKE	Isogeny-based (broken)	PKE/KEM	Candidate, eliminated

Table 2.3 Remaining candidate algorithms from the NIST PQC Fourth Round (PKE/KEM)

After several years of evaluation by NIST, the selected algorithms are based on different mathematical approaches that are considered secure against quantum

attacks and that satisfy the modern security notions used for public-key encryption and key-encapsulation.
In particular, NIST evaluates candidates with respect to indistinguishability under adaptive chosen-ciphertext attacks (IND-CCA2). However, this requirement is relaxed a little, as submitted schemes will be evaluated based on how well they appear to provide this property, but the proof of security is not strictly required. Furthermore, for a subset of applications like purely ephemeral key exchange, indistinguishability under chosen-plaintext attacks (IND-CPA) may be sufficient [10].

Although lattice-based schemes make up most of the first standardized algorithms due to their good performance and solid security analysis within these models, code-based cryptography remains an important and well-established alternative.

This thesis focuses specifically on code-based public-key encryption. Among the code-based candidates, two stand out for their historical significance and contrasting design choices: the McEliece cryptosystem and HQC (Hamming Quasi-Cyclic). McEliece, introduced in 1978, is one of the oldest post-quantum proposals and relies on the hardness of decoding random-looking binary Goppa codes. In contrast, HQC is a more recent system selected by NIST in the fourth round. It employs the use of quasi-cyclic codes to reduce key sizes compared to unstructured random codes and it implements a "double encryption" structure where a message bit is encoded using a simple repetition code, then concealed by two independent random vectors.

Comparing these two systems offers insight into the evolution of code-based cryptography: McEliece represents the classic “hide-a-structure” paradigm with exceptional long-term security confidence but larger keys, whereas HQC exemplifies a modern approach that trades some structural simplicity for better efficiency and a direct security reduction.
As for the coding-theoretic background in Chapter 1, the presentation of the McEliece and HQC cryptosystems in this chapter is inspired by [1].

2.2 McEliece Cryptosystem

In 1978, R. J. McEliece [11] proposed a new public key cryptosystem based on the theory of algebraic codes, now called the McEliece cryptosystem. The framework is

structured in three phases: key generation, encryption and decryption. The scheme considers the usual actors Alice, Bob and Eve.

- **Key Generation**. In the initial phase, Alice chooses a generator matrix G of a $[n,k]$-q-ary Goppa code capable of correcting t errors. The matrix G represents the structured code, which cannot be made public as it is the key to decode the ciphertext. Instead, Alice makes public a scrambled version of G, obtained by multiplying the generator matrix with a random non-singular $k \times k$ matrix S and a $n \times n$ permutation matrix P. The new generator matrix is then computed as $G' = SGP$. In this way the code generated by G' appears random.
  The pair (G',t) is the public key, while the triple (S,G,P) is the private key;

- **Encryption**. Let $\mathbf{m} \in \mathbb{F}_q^k$ be the message that Bob wants to transmit and $\mathbf{e} \in \mathbb{F}_q^n$ a random vector with Hamming weight t. To encrypt the plaintext Bob computes the ciphertext $\mathbf{c} = \mathbf{m}G' + \mathbf{e}$, which can be interpreted as adding t errors to the $\mathbf{m}G'$ codeword.
  An eavesdropper, Eve, only knows the public pair (G',t) and the ciphertext $\mathbf{c}$. To retrieve the message $\mathbf{m}$, she would need to decrypt the code generated by G', which appears random to her. Thus, she is facing an NP-complete problem and the best known solvers have an exponential cost.
  The other option is to recover the private key (S,G,P) from the public key, however the code equivalence problem is hard;

- **Decryption**. On the other hand, Alice own the private key (S,G,P), so she can easily recover $\mathbf{m}$. The first step consists in evaluating $\mathbf{c}' = \mathbf{c}P^{-1}$, an intermediate product that retains the same Hamming weight as the original ciphertext. Then Alice can implement a decoding algorithm to correct t errors in $\mathbf{c}'$, obtaining another intermediate product $\mathbf{m}'$. The final step is to compute $\mathbf{m}'S^{-1}$, which equals the original message $\mathbf{m}$.
  An example of error correction algorithm for Goppa codes is the Patterson algorithm.

The McEliece encryption scheme is summarized in Table 2.4.

The proof of correctness is straightforward, and proceeds as follows. By observing the encrypted message $\mathbf{y} = \mathbf{m}\hat{G} + \mathbf{e}$ and the public generator matrix $G' = SGP$,

ALICE		BOB
KEY GENERATION		
Choose a linear code $\mathscr{C} \subseteq \mathbb{F}_q^n$ of dimension k and error correction capacity t. Let G be a $k \times n$ generator matrix of a chosen Goppa code. Choose a randomly non-singular $k \times k$ matrix S and an $n \times n$ permutation matrix P. Compute $G' = SGP$. The public key is given by $\mathsf{pk} = (t, G')$ and the secret key by $\mathsf{sk} = (G, S, P)$	$\xrightarrow{\mathsf{pk}}$	
		ENCRYPTION
		Choose a message $\mathbf{m} \in \mathbb{F}_q^k$ and a random error vector $\mathbf{e} \in \mathbb{F}_q^n$ of weight at most t Encrypt the message $\mathbf{c} = \mathbf{m}G' + \mathbf{e}$
	$\xleftarrow{\mathbf{c}}$	
DECRYPTION		
Decrypt the cipher, by decoding $\mathbf{c}P^{-1} = \mathbf{m}SG + \mathbf{e}P^{-1}$ to get $\mathbf{m}S$, and finally recover the message as $\mathbf{m} = (\mathbf{m}S)S^{-1}$		

Table 2.4 McEliece Framework

and evaluating $\mathbf{y}' = \mathbf{y}P^{-1}$, the result is

$$(\mathbf{m}\hat{G} + \mathbf{e})P^{-1} = (\mathbf{m}S)G + \mathbf{e}P^{-1},$$

where $\mathbf{e}P^{-1}$ is still a vector in $\mathbb{F}_q^n$ with Hamming weight t. The implementation of the error correction algorithm to $\mathbf{y}'$ removes the error $\mathbf{e}P^{-1}$, obtaining $\mathbf{m}' = (\mathbf{m}S)$. Thus,

$$\mathbf{m}'S^{-1} = (\mathbf{m}S)S^{-1} = \mathbf{m}.$$

The cryptosystem can be defined in a more mathematical formal way, according to the model introduced in the previous chapter.
Let $\mathbb{F}_q$ be a finite field of order q, and let $\mathscr{C}$ be a family of t-error-correcting $[n,k]$-linear codes over $\mathbb{F}_q$. Let $\mathsf{Decode}_{\mathscr{C}}(\cdot)$ be an efficient decoding algorithm for codes in $\mathscr{C}$, capable of correcting up to t errors. Let $\mathsf{GenCode}(1^\lambda)$ be a randomized algorithm that outputs a description of a code $\Gamma \in \mathscr{C}$ with parameters $[n,k,t]$, and let $G \in \mathbb{F}_q^{k\times n}$ be a generator matrix for Γ. Let $\mathbf{m} \in \mathbb{F}_q^k$ be the plaintext and $\mathbf{c} \in \mathbb{F}_q^n$ the ciphertext.

The McEliece public-key encryption scheme is defined as follows:

- **Key Generation**: $\mathsf{KeyGen}(1^\lambda) \to (\mathsf{pk}, \mathsf{sk})$
  1. Generate a t-error-correcting $[n,k]$-linear code Γ over $\mathbb{F}_q$: $\Gamma \leftarrow \mathsf{GenCode}(1^\lambda)$ and let $G \in \mathbb{F}_q^{k\times n}$ be a generator matrix for Γ;
  2. Choose uniformly at random:
     - An invertible matrix $S \xleftarrow{\$} \mathsf{GL}_k(\mathbb{F}_q)$;
     - A permutation matrix $P \xleftarrow{\$} \mathsf{Perm}_n(\mathbb{F}_q)$.
  3. Compute the scrambled generator matrix: $G' \leftarrow SGP$;
  4. Output: $\mathsf{pk} \leftarrow (G', t), \mathsf{sk} \leftarrow (S, G, P)$.
- **Encryption**: $\mathsf{Enc}_{\mathsf{pk}}(\mathbf{m}) \to \mathbf{c}$
  1. Choose a random error vector $\mathbf{e} \xleftarrow{\$} \mathbb{F}_q^n$ with Hamming weight $wt_H(\mathbf{e}) = t$;
  2. Compute the ciphertext: $\mathbf{c} \leftarrow \mathbf{m}G' + \mathbf{e}$;
  3. Output $\mathbf{c}$.
- **Decryption** $\mathsf{Dec}_{\mathsf{pk}}(\mathbf{c}) \to \mathbf{m}$ or $\perp$
  1. Apply the inverse permutation: $\mathbf{c}' \leftarrow \mathbf{c}P^{-1}$;
  2. Decode $\mathbf{c}'$ using the structured code Γ: $\mathbf{m}' \leftarrow \mathsf{Decode}_{\mathscr{C}}(\mathbf{c}')$. If decoding fails, output $\perp$;
  3. Recover the plaintext $\mathbf{m} \leftarrow \mathbf{m}'S^{-1}$;
  4. Output $\mathbf{m}$.

2.2.1 McEliece Framework for Key Encapsulation Mechanisms

The Classic McEliece Key Encapsulation Mechanism (KEM) is a post-quantum cryptographic primitive submitted to the NIST Post-Quantum Cryptography standardization process. It builds upon the original McEliece public-key encryption scheme by combining it with the Fujisaki–Okamoto (FO) transform to obtain a KEM that is IND-CCA2 secure in the random oracle model, assuming the hardness of decoding random linear codes.

The core idea of the transation between McEliece-PKE to McEliece-KEM is to encrypt a random error $\mathbf{e}$ instead of an arbitrary message $\mathbf{m}$, thus, the error vector itself is the message: the sender generates $\mathbf{e}$ pseudo-randomly from a seed and then re-derives both the ciphertext and the session key deterministically. The receiver, after decoding, re-encrypts the recovered $\mathbf{e}$ and accepts only if the re-encryption matches the received ciphertext. This re-encryption check is the mechanism that defeats chosen-ciphertext attacks.

Classic McEliece uses the implicit rejection variant of FO (sometimes called $\mathsf{FO}^{\not\perp}$ or the Dent/Hofheinz–Hövelmanns–Kiltz variant). On decapsulation failure, rather than returning $\perp$ explicitly, the decapsulation algorithm outputs a pseudorandom session key derived from a secret string s stored in the private key. This prevents an adversary from distinguishing decapsulation failures from successes, which is essential for CCA2 security.

The Classic McEliece KEM is built on binary Goppa codes, and it is parameterised by integers (m,n,t), where $q = 2^m$, giving a code of length n, dimension $k = n - mt$, and error-correction capability t. All selected parameter sets target $\ell = 256$ bits of security.

Let $\mathscr{H}$ be a cryptographic hash function modeled as a random oracle, outputting $\ell = 256$ bits, and let $\mathscr{G}$ be a pseudorandom bit generator used to expand a short seed into all key-generation material. An additional secret string $\mathbf{s} \in \mathbb{F}_2^n$, extracted from the seed expansion $\mathscr{G}(\delta)$ alongside the Goppa code parameters, is stored in the private key; it is used to derive a pseudorandom fallback session key $\bar{\mathsf{K}}$ upon decapsulation failure, implementing implicit rejection.

The three algorithms constituting Classic McEliece KEM are described below [12].

- **Key Generation:** $\mathsf{KeyGen}(1^\lambda) \to (\mathsf{pk}, \mathsf{sk})$

  1. Sample a random seed $\delta \xleftarrow{\$} \{0,1\}^\ell$ and expand it via $\mathscr{G}$ to derive a random binary Goppa code $\Gamma = (g, \alpha_0, \ldots, \alpha_{n-1})$, where $g \in \mathbb{F}_q[x]$ is an irreducible polynomial of degree t and $\alpha_0, \ldots, \alpha_{n-1}$ are distinct elements of $\mathbb{F}_q$. If any derivation step fails, refresh the seed and restart;
  2. Compute the parity-check matrix of Γ in systematic form $(I_{mt} \mid T)$ and set the public key $\mathsf{pk} \leftarrow T$;
  3. Output $\mathsf{pk} \leftarrow T, \quad \mathsf{sk} \leftarrow (\delta, g, \alpha, \mathbf{s})$, where $\mathbf{s} \in \mathbb{F}_2^n$ is extracted from the seed expansion $\mathscr{G}(\delta)$ alongside the code parameters.

- **Encapsulation:** $\mathsf{Encaps}(\mathsf{pk}) \to (\mathbf{c}, \mathsf{K})$

  1. Sample a uniformly random vector $\mathbf{e} \xleftarrow{\$} \mathbb{F}_2^n$ of Hamming weight t;
  2. Compute the ciphertext as the syndrome of $\mathbf{e}$ under the public parity-check matrix: $\mathbf{c} \leftarrow (I_{mt} \mid T)\,\mathbf{e} \in \mathbb{F}_2^{mt}$;
  3. Derive the session key: $\mathsf{K} \leftarrow H(1, \mathbf{e}, \mathbf{c})$;
  4. Output the ciphertext $\mathbf{c}$ and session key K.

- **Decapsulation:** $\mathsf{Decaps}(\mathsf{sk}, \mathbf{c}) \to \mathsf{K}$

  1. Using the private Goppa code Γ, attempt to decode $\mathbf{c}$: find a weight-t vector $\mathbf{e}' \in \mathbb{F}_2^n$ such that $(I_{mt} \mid T)\,\mathbf{e}' = \mathbf{c}$;
  2. If decoding succeeds, output $\mathsf{K} \leftarrow \mathscr{H}(1, \mathbf{e}', \mathbf{c})$;
  3. If decoding fails (no such $\mathbf{e}'$ exists), apply implicit rejection: output the pseudorandom fallback key $\bar{\mathsf{K}} \leftarrow \mathscr{H}(0, \mathbf{s}, \mathbf{c})$.

In both the success and failure branches of decapsulation, the output is a well-defined pseudorandom value. An adversary submitting a malformed ciphertext therefore receives a response that is computationally indistinguishable from a valid session key, which is the defining property of the $\mathsf{FO}^{\not\perp}$ transform and the basis of the IND-CCA2 security reduction.

2.2.2 Classic McEliece Parameters

The National Institute of Standards and Technology (NIST) has established a standardized framework for categorizing the security strength of cryptographic algorithms in the post-quantum era. These security levels provide a uniform basis for comparing different cryptographic schemes and ensure consistent protection across various applications. The NIST security levels are defined as follows:

- **Level 1**: 128-bit security (comparable to AES-128);
- **Level 3**: 192-bit security (comparable to AES-192);
- **Level 5**: 256-bit security (comparable to AES-256).

Table 2.5 summarizes five proposed parameter sets of McEliece-KEM across these security levels. The table includes the fundamental code parameters (m,n,t), the resulting public and private key sizes in bytes, the ciphertext size, and the corresponding NIST security level.

Instance	Security	m	n	t	pk	sk	**c**
mceliece348864	Level 1	12	3488	64	261120	6492	96
mceliece460896	Level 3	13	4608	96	524160	13608	156
mceliece6688128	Level 5	13	6688	128	1044992	13932	208
mceliece6960119	Level 5	13	6960	119	1047319	13948	194
mceliece8192128	Level 5	13	8192	128	1357824	14120	208

Table 2.5 Parameters for Classic McEliece

Standard and "f" parameter sets.

For each security level, Classic McEliece defines two variants of the parameter set: a standard one and a corresponding f variant (e.g., mceliece6688128 and mceliece6688128f). Both use identical code parameters n, t, and m, and are functionally indistinguishable during encapsulation and decapsulation; the difference lies entirely in key generation. The standard variant requires the parity-check matrix to

be reduced to systematic form, however this succeeds only for a subset of candidate codes (approximately 29% of attempts), so key generation must often be restarted with a new seed. The f variant instead accepts the more general semi-systematic form, which admits a strictly larger set of valid matrices while preserving an equivalent security reduction. This relaxation reduces the failure probability of each key-generation attempt to below 2^{-30}, so that in practice a single attempt almost always succeeds, yielding faster and less variable key-generation times at negligible cost to security (at most a 2-bit tightness loss in the underlying proof) [12].

2.3 HQC Cryptosystem

Hamming Quasi-Cyclic (HQC) is another cryptographic framework based on the theory of algebraic codes, however it relies on a different structure and different codes with respect to McEliece. Indeed, the cryptosystem uses a probabilistic approach in the encryption schemes and does not hide the initial code, which needs to be efficiently decodable. The message gets encrypted as codeword to which an error, too large to decode, gets added. With the knowledge of the private key parts of this error can be cancelled out resulting (with high probability) in a vector which can be decoded to recover the message. Furthermore, the HQC exploits two different codes: An $[n,k]$-linear code $\mathscr{C}$ over $\mathbb{F}_q$, which serves as error corrector since can efficiently decode t errors, and a random quasi-cyclic $[2n,n]$-code presented through a parity-check matrix

$$H = \begin{pmatrix} I_n & | \operatorname{rot}(\mathbf{h}) \end{pmatrix},$$

which does not require to be efficiently decodable, as it has security purposes.

In order to explain the algorithm, some parameters need to be introduced.
Let n be a positive integer, q be a prime power and $\mathscr{R} = \mathbb{F}_q[x]/(x^n - 1)$ be a polynomial rings over a finite field.
Elements of $\mathbb{F}_q^n$ correspond naturally to polynomials in $\mathscr{R}$ via the linear map

$$\mathbf{a} = (a_0, a_1, \ldots, a_{n-1}) \longleftrightarrow a(x) = a_0 + a_1 x + \cdots + a_{n-1} x^{n-1}.$$

Multiplication in $\mathscr{R}$ corresponds to cyclic convolution in $\mathbb{F}_q^n$.
Let w, w_r, and w_e be positive integers of order $\sqrt{n}/2$. These are public parameters controlling the weight of secret and error vectors.

The cryptosystem can be divided into three steps:

1. **Key generation**. Based on the public parameters (n,k,t,w,w_r,w_e) Alice chooses a random generator $G \in \mathbb{F}_q^{k\times n}$ for a linear code $\mathscr{C}$, a random public polynomial $h(x) \in \mathscr{R}$ and two secret elements $y(x),z(x) \in \mathscr{R}$, each of Hamming weight w.
   The pair $(y(x),z(x))$ represents the private key, while the public key is composed by the generator matrix G, the random element $h(x)$ and $s(x) = y(x) + h(x)z(x)$;

2. **Encryption**. Let $\mathbf{m} \in \mathbb{F}_q^k$ be the message that Bob wants to transmit to Alice. In order to encrypt it, he chooses a random polynomial $e(x) \in \mathscr{R}$ of Hamming weight w_e and two random polynomials $r_1(x),r_2(x) \in \mathscr{R}$, both of Hamming weight w_r and then evaluates the two equalities

$$\begin{aligned} u(x) &= r_1(x) + h(x)r_2(x); \\ v(x) &= m'(x) + s(x)r_2(x) + e(x), \end{aligned}$$

   where $m'(x) \in \mathscr{R}$ is the polynomial corresponding to $\mathbf{m}G \in \mathbb{F}_q^n$.
   The cipher is then given by $\mathbf{c} = (u(x),v(x))$.
   The message $\mathbf{m}$ is thus encoded through the public matrix G and perturbed through the error vector $s(x)r_2(x) + e(x)$. It can be noticed that both $r_2(x)$ and $e(x)$ are randomly chosen by Bob, while Alice is only able to control $s(x)$ through her secret. Decryption is made possible by this secret and the additional structured information $u(x)$ that Bob provides regarding $r_2(x)$;

3. **Decryption**. In the decryption phase Alice recovers the original message $\mathbf{m}$ by applying the error correction algorithm on the result of the operation $v(x) - u(x)z(x)$, since

$$\begin{aligned} v(x) - u(x)z(x) &= m'(x) + s(x)r_2(x) + e(x) - [r_1(x) + h(x)r_2(x)]z(x) \\ &= m'(x) + [y(x) + h(x)z(x)]r_2(x) + e(x) - r_1(x)z(x) - h(x)r_2(x)z(x) \\ &= m'(x) + y(x)r_2(x) - r_1(x)z(x) + e(x) \end{aligned}$$

   The equality shows that the term $(y(x)r_2(x) - r_1(x)z(x) + e(x))$ is the total error added to the codeword $\mathbf{m}G$, so the decryption will succeed if and only if the Hamming weight of this error is less or equal to t. This condition is

satisfied with high probability, hence the decryption has a great likelihood of succeeding.

Note that all polynomials of the framework corresponds to vectors in $\mathbb{F}_q^n$.
The HQC encryption scheme is summarized in Table 2.6.

For an eventual eavesdropper Eve it is impossible to recover the private elements $y(x)$ and $z(x)$ from the public information $s(x)$ and $h(x)$. In fact, $s(x)$ can be written as

$$S = (\mathbf{y}, \mathbf{z}) \begin{pmatrix} I_n \\ \text{rot}(\mathbf{h}) \end{pmatrix},$$

thus $H = (I_n, \text{rot}(\mathbf{h})^\top)$ acts as quasi-cyclic parity-check matrix and $(\mathbf{y}, \mathbf{z})$ as unknown error vector.

This framework can safely publish the generator matrix G for two primary reasons:

1. **Code Randomization**. The underlying algebraic code is randomized by the secret parameter $h(x)$, which determines the parity-check matrix H. The public sees only G, not the specific H used for syndrome calculations;
2. **Excessive Error Weight**. The error vector $s(x)r_2(x) + e(x)$ added during encryption intentionally has a Hamming weight greater than the code's public error-correction capacity t. This renders the efficient decoder useless for an attacker.

In fact, $\mathbf{u}$ and $\mathbf{s}$ are syndromes under H of low-weight vectors:

$$\mathbf{u} = (\mathbf{r}_1, \mathbf{r}_2)H^\top \quad \text{and} \quad \mathbf{s} = (\mathbf{y}, \mathbf{z})H^\top.$$

Recovering $(r_1(x), r_2(x))$ or $(y(x), z(x))$ from these syndromes requires solving the NP-hard syndrome decoding problem. Moreover, since $\text{wt}_H(s(x)r_2(x) + e(x)) > t$, knowledge of G and the ciphertext $\mathbf{v}$ does not allow an attacker to uniquely determine the message $\mathbf{m}$.

Since the algebraic code, which is efficiently decodable, is publicly known, the security of this framework is different to that of the McEliece framework, as it does not rely on the indistinguishability of the code.

ALICE		BOB
KEY GENERATION		
Choose an $[n,k]$-linear code $\mathscr{C}$ over $\mathbb{F}_q$, which can efficiently decode t errors with generator matrix $G \in \mathbb{F}_q^{k\times n}$ and choose $h(x) \xleftarrow{\$} \mathscr{R}$ Choose $y(x), z(x) \xleftarrow{\$} \mathscr{R}$ such that $wt_H(y(x)) = wt_H(z(x)) = w$ Compute $s(x) \leftarrow y(x) + h(x)z(x)$ The public key is $\mathsf{pk} = (G, h(x), s(x), w_e, w_r)$ and the secret key is $\mathsf{sk} = (y(x), z(x))$		
	$\xrightarrow{\mathsf{pk}}$	
		ENCRYPTION
		Choose a message $\mathbf{m} \in \mathbb{F}_q^k$ Compute the codeword $\mathbf{m}G$ and the corresponding polynomial $m'(x)$ Choose $e(x) \xleftarrow{\$} \mathscr{R}$ such that $wt_H(e(x)) = w_e$ Choose $r_1(x), r_2(x) \xleftarrow{\$} \mathscr{R}$ such that $wt_H(r_1(x)) = wt_H(r_2(x)) = w_r$ Compute $u(x) = r_1(x) + h(x)r_2(x)$ Compute $v(x) = m'(x) + s(x)r_2(x) + e(x)$ The cipher is $\mathbf{c} = (u(x), v(x))$
	$\xleftarrow{\mathbf{c}}$	
DECRYPTION		
Compute $c'(x) \leftarrow v(x) - u(x)z(x)$ and use the decoding algorithm of $\mathscr{C}$ to recover $\mathbf{m}$		

Table 2.6 HQC Framework

The cryptosystem can be defined in a more mathematical formal way, according to the model introduced in the previous chapter.
Let $\mathbb{F}_q$ be a finite field of order q, and let $\mathscr{C}$ be a family of t-error-correcting $[n,k]$-

linear codes over $\mathbb{F}_q$. Let $\text{Decode}_{\mathscr{C}}(\cdot)$ be an efficient decoding algorithm for codes in $\mathscr{C}$, capable of correcting up to t errors. Let $\mathscr{R} = \mathbb{F}_q[x]/(x^n - 1)$ be the polynomial ring where multiplication corresponds to cyclic convolution.
Let the public parameters be:

- n, k, t: parameters of the linear code $\mathscr{C}$;
- w, w_r, w_e: positive integers of order $\sqrt{n}/2$ controlling the weights of various polynomials.

The HQC public-key encryption scheme is defined as follows:

- **Key Generation**. $\mathsf{KeyGen}(1^\lambda) \to (\mathsf{pk}, \mathsf{sk})$
  1. Generate a t-error-correcting $[n,k]$-linear code $\mathscr{C}$ over $\mathbb{F}_q$, with generator matrix $G \in \mathbb{F}_q^{k \times n}$;
  2. Choose uniformly at random:
     - A public polynomial $h(x) \xleftarrow{\$} \mathscr{R}$;
     - Two secret polynomials $y(x), z(x) \xleftarrow{\$} \mathscr{R}$, each of Hamming weight w.
  3. Compute the public syndrome: $s(x) \leftarrow y(x) + h(x)z(x)$;
  4. Output: $\mathsf{pk} \leftarrow (G, h(x), s(x), w_e, w_r)$, $\quad \mathsf{sk} \leftarrow (y(x), z(x))$.
- **Encryption**: $\mathsf{Enc}_{\mathsf{pk}}(\mathbf{m}) \to \mathbf{c}$
  1. Encode the plaintext $\mathbf{m} \in \mathbb{F}_q^k$ into a codeword $\mathbf{m}G$, and derive the corresponding polynomial $m'(x)$;
  2. Choose uniformly at random:
     - Two polynomials $r_1(x), r_2(x) \xleftarrow{\$} \mathscr{R}$, each of Hamming weight w_r;
     - An error polynomial $e(x) \xleftarrow{\$} \mathscr{R}$ of Hamming weight w_e.
  3. Compute:
     $$u(x) \leftarrow r_1(x) + h(x)r_2(x),$$
     $$v(x) \leftarrow m'(x) + s(x)r_2(x) + e(x);$$
  4. Output the ciphertext: $\mathbf{c} \leftarrow (u(x), v(x))$.

- **Decryption**: $\mathsf{Dec}_{\mathsf{sk}}(\mathbf{c}) \rightarrow \mathbf{m}$ or $\perp$

  1. Compute: $c'(x) \leftarrow v(x) - u(x)z(x)$;
  2. Decode $c'(x)$ using the error-correcting code $\mathscr{C}$: $\mathbf{m} \leftarrow \mathsf{Decode}_{\mathscr{C}}(c'(x))$;
     - If decoding fails (i.e., $wt_H(y(x)r_2(x) - r_1(x)z(x) + e(x)) > t$), output $\perp$;
     - If decoding succeeds, output $\mathbf{m}$.

2.3.1 HQC Framework for Key Encapsulation Mechanisms

The HQC Public-Key Encryption scheme described above can be transformed into a Key Encapsulation Mechanism (KEM), which constitutes its main practical application.
The KEM construction builds upon the underlying PKE scheme, but adapts it to encapsulate a symmetric key rather than encrypt an arbitrary message. The resulting HQC-KEM is defined by the triple of algorithms $(\mathsf{KeyGen}, \mathsf{Encaps}, \mathsf{Decaps})$, derived from the PKE algorithms $(\mathsf{KeyGen}, \mathsf{Enc}, \mathsf{Dec})$.

HQC-KEM is obtained by applying a (salted) Fujisaki–Okamoto transformation with implicit rejection to HQC-PKE.
In this construction, randomness used in the PKE encryption algorithm is no longer sampled uniformly, but deterministically derived from hashed inputs, ensuring consistency during decapsulation.

More precisely, let $\mathscr{G}$, $\mathscr{H}$ and $\mathscr{J}$ be cryptographic hash functions, modeled as random oracles. Their roles are as follows:

- $\mathscr{G} : \{0,1\}^* \rightarrow \{0,1\}^{|\mathsf{K}|} \times \mathsf{COINS}^{\mathsf{asym}}$ takes as input a hash of the public key together with a message $\mathbf{m}$ and a random salt, and outputs both the encapsulated key K and the deterministic randomness θ used in HQC-PKE encryption;
- $\mathscr{H} : \{0,1\}^* \rightarrow \{0,1\}^*$ is used to hash the public key and contributes to domain separation and key derivation;
- $\mathscr{J} : \{0,1\}^* \rightarrow \{0,1\}^{|\mathsf{K}|}$ takes as input a hash of the public key together with a random secret value σ and the ciphertext $\mathbf{c}$, and outputs the rejection key $\bar{\mathsf{K}}$.

The pseudorandom fallback key $\bar{\mathsf{K}}$ derived during decapsulation is used in case of decryption or verification failure, ensuring implicit rejection and resistance against chosen-ciphertext attacks.
Finally, let θ denote the deterministic randomness used in the HQC-PKE encryption algorithm, derived from $\mathscr{G}$.

The three algorithms constituting HQC-KEM are described below [13].

- **Key Generation:** $\mathsf{KeyGen}(1^\lambda) \to (\mathsf{ek}, \mathsf{dk})$
  1. Generate HQC-PKE keys $(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{KeyGen}_{\mathsf{PKE}}(1^\lambda)$;
  2. Sample a secret value $\sigma \xleftarrow{\$} \{0,1\}^\lambda$;
  3. Output the keys $\mathsf{ek} \leftarrow \mathsf{pk}, \quad \mathsf{dk} \leftarrow (\mathsf{sk}, \mathsf{pk}, \sigma)$.
- **Encapsulation:** $\mathsf{Encaps}(\mathsf{ek}) \to (\mathbf{c}, \mathsf{K})$
  1. Sample $\mathbf{m} \xleftarrow{\$} \mathbb{F}_q^k, \quad \mathrm{salt} \xleftarrow{\$} \{0,1\}^\lambda$;
  2. Derive key and randomness $(\mathsf{K}, \theta) \leftarrow \mathscr{G}(\mathscr{H}(\mathsf{ek}) \,\|\, \mathbf{m} \,\|\, \mathrm{salt})$;
  3. Compute ciphertext using HQC-PKE $\mathbf{c}_{\mathsf{PKE}} \leftarrow \mathsf{Enc}_{\mathsf{PKE}}(\mathsf{ek}, \mathbf{m}; \theta)$;
  4. Output $\mathbf{c} \leftarrow (\mathbf{c}_{\mathsf{PKE}}, \mathrm{salt}), \quad \mathsf{K}$.
- **Decapsulation:** $\mathsf{Decaps}(\mathsf{dk}, \mathbf{c}) \to \mathsf{K}$
  1. Parse $\mathsf{dk} = (\mathsf{sk}, \mathsf{pk}, \sigma), \quad \mathbf{c} = (\mathbf{c}_{\mathsf{PKE}}, \mathrm{salt})$;
  2. Decrypt $\mathbf{m}' \leftarrow \mathsf{Dec}_{\mathsf{PKE}}(\mathsf{sk}, \mathbf{c}_{\mathsf{PKE}})$;
  3. Recompute $(\mathsf{K}', \theta') \leftarrow \mathscr{G}(\mathscr{H}(\mathsf{pk}) \,\|\, \mathbf{m}' \,\|\, \mathrm{salt})$;
  4. Re-encrypt $\mathbf{c}'_{\mathsf{PKE}} \leftarrow \mathsf{Enc}_{\mathsf{PKE}}(\mathsf{pk}, \mathbf{m}'; \theta')$;
  5. Compute fallback key: $\bar{\mathsf{K}} \leftarrow \mathscr{J}(\mathscr{H}(\mathsf{pk}) \,\|\, \sigma \,\|\, \mathbf{c})$;
  6. If $\mathbf{m}' = \perp$ or $\mathbf{c}'_{\mathsf{PKE}} \neq \mathbf{c}_{\mathsf{PKE}}$, output $\bar{\mathsf{K}}$;
  7. Otherwise, output K.

2.3.2 HQC Parameters

HQC is typically instantiated at three security levels, aligned with the NIST categories previously described.

Each level uses different code parameters, resulting in larger keys and ciphertexts for higher security. The following Table 2.7 contains the proposed parameters for HQC-PKE together with an upper estimate on the decoding failure rate (DFR) and ciphertext size and key sizes (given in bytes).

Instance	Security	n	w	$w_r = w_e$	pk	sk	**c**	DFR
HQC-1	Level 1	17669	66	75	2249	40	4481	2^{-128}
HQC-3	Level 3	35851	100	114	4522	40	9026	2^{-192}
HQC-5	Level 5	57637	131	149	7245	40	14469	2^{-256}

Table 2.7 Parameters for HQC-PKE [13]

On the other hand, Table 2.8 reports the encapsulation key, decapsulation key, ciphertext and shared key size (in byets) corresponding to each security level. Furthermore, the salt has constant size of 32 B.

Instance	Security	$\|\mathsf{ek}\|$	$\|\mathsf{dk}\|$	$\|\mathsf{c}\|$	$\|\mathsf{K}\|$
HQC-1	Level 1	2241	2321	4433	32
HQC-3	Level 3	4514	4602	8978	32
HQC-5	Level 5	7237	7333	14421	32

Table 2.8 Parameters for HQC-KEM [13]

Chapter 3

Benchmark of McEliece and HQC Key Encapsulation Mechanisms

3.1 Motivations and Objectives of the Benchmark

The theoretical treatment developed in the previous chapters has characterized McEliece and HQC as two structurally distinct code-based Key Encapsulation Mechanisms selected within the NIST Post-Quantum Cryptography standardization process.
Although the security properties and algebraic structures of the two schemes have already been discussed, a theoretical comparison alone is not sufficient to guide practical engineering decisions. Cryptographic primitives that provide the same level of security can still differ significantly in their computational and memory requirements when implemented in real systems. This difference between theoretical security equivalence and practical performance motivates the empirical evaluation presented in this chapter.

The central objective of the benchmark presented in this chapter is therefore to quantify, on a common hardware and software baseline, the practical performance profile of the two schemes across the three canonical KEM operations: key-pair generation, encapsulation, and decapsulation. In addition to measuring execution time, the benchmark examines several other factors that are important for real-world deployment. These include the size of the public key, private key, and ciphertext; the peak resident memory usage; the number of CPU cycles required for each

operation, used as a hardware-independent measure of computational cost; the statistical stability of the results, expressed through the coefficient of variation.

A second, important motivation for the design of this benchmark lies in the reproducibility and statistical soundness of the measurements. Microbenchmarking cryptographic primitives on general-purpose operating systems is vulnerable to a wide range of confounding factors: scheduler-induced jitter, non-deterministic cache and TLB states, frequency scaling, background system load, and memory-allocator behavior can all introduce noise that, if not controlled for, can easily dwarf the actual difference between two competing implementations. Consequently, a second explicit goal of the benchmark is to establish an experimental protocol robust enough to yield statistically defensible conclusions and to make explicit, through confidence intervals and significance indicators, whether an observed difference between McEliece and HQC is large enough to be considered a genuine algorithmic effect or whether it falls within the expected noise floor of the measurement environment.

3.2 Benchmark Environment and Experimental Methodology

3.2.1 Reference Implementations and Parameter Sets

Choice of Framework and Implementation Language

The benchmark is conducted using the PQClean [14] framework as the source of reference implementations for both Classic McEliece and HQC. PQClean is a collaborative project that collects clean, portable, and tested implementations of post-quantum cryptographic schemes from the NIST standardization process. PQClean enforces a strict set of engineering conventions, like namespace isolation, absence of dynamic memory allocation, constant-time execution where applicable, and a uniform API across schemes, which facilitates a fair comparison between different algorithms and ensures the reproducibility of the thesis.

The choice of the C programming language is motivated by both performance and compatibility considerations. C is the language used by the PQClean reference implementations and remains the de facto standard for low-level cryptographic

software. Its low-level control over memory layout and execution timing makes it particularly well-suited for evaluating side-channel resistance and computational efficiency, two central concerns in post-quantum cryptography research.

Parameter Sets Selection for Benchmarking

To conduct a meaningful performance comparison between Classic McEliece and HQC, it is essential to compare only those parameter sets that target the same NIST security level. This ensures that performance differences observed in the benchmark are attributable to algorithmic design choices rather than variations in security strength.
The NIST Post-Quantum Cryptography standardization process defines five security categories, where Level 1 corresponds to AES-128 security and Level 5 corresponds to AES-256 security. For this benchmark, we include all relevant security levels to provide a comprehensive performance assessment across the entire spectrum of NIST-defined security categories.

At each security level, Classic McEliece offers multiple parameter sets, while HQC typically provides a single optimized parameter set per level. Table 3.1 summarizes all parameter sets included in this benchmark, organized by their respective NIST security level.

Classic McEliece offers multiple variants at each security level: the standard variants (e.g., mceliece348864, mceliece460896) and the "f" variants (mceliece348864f, mceliece460896f). The "f" variants are optimized for faster key generation: instead of requiring the public-key matrix to be reduced to strict systematic form, they accept the more general semi-systematic form, which succeeds on the first attempt with much higher probability. This reduces the number of restarts needed during key generation, at the cost of a marginally looser security proof (at most a 2-bit tightness loss), while encapsulation and decapsulation remain identical to the standard variants.
At Level 5, three distinct variants exist (mceliece6688128, mceliece6960119, and mceliece8192128), each representing different trade-offs between key size, ciphertext size, and computational speed. Including all variants provides a complete picture of the trade-offs available within the McEliece family.

Security Level	Classic McEliece	HQC
1	mceliece348864 mceliece348864f	hqc-128
3	mceliece460896 mceliece460896f	hqc-192
5	mceliece6688128 mceliece6688128f mceliece6960119 mceliece6960119f mceliece8192128 mceliece8192128f	hqc-256

Table 3.1 Parameter sets for NIST security levels

For HQC, each security level has a single recommended parameter set (hqc-128, hqc-192, and hqc-256). Testing all three ensures that the comparison covers the entire range of NIST security categories relevant to real-world deployments.

3.2.2 Control of Environmental Noise

To limit the influence of the operating system scheduler and of contention with other processes, the harness pins the benchmarking process to a single logical CPU core (core 0) via sched_setaffinity, and attempts to raise the scheduling priority of the process to the maximum value permitted by setpriority (an operation that requires elevated privileges to take full effect, and which the harness treats as best-effort, falling back gracefully to the default priority when it is denied).
In addition, prior to any timed measurement, the harness executes a CPU warm-up routine, warmup_cpu, consisting of a tight loop of floating-point square-root operations, whose purpose is to bring the processor out of any low-power or reduced-frequency idle state and into a stable operating-frequency regime before the first timed iteration begins. An algorithm-specific warm-up is also performed by warmup_algorithm_mceliece and warmup_algorithm_hqc, consisting of a small number of untimed key-generation calls (WARMUP_ITERATIONS $= 5$), so that any

one-time, iteration-independent cost, such as the initial population of code caches, the first-touch allocation of memory pages, or lazy library initialization, is absorbed before the timed measurements start, rather than inflating the first recorded sample.

A further and architecturally significant methodological choice is that each algorithm is benchmarked inside its own isolated child process, spawned through fork() and synchronized with the parent through a unidirectional POSIX pipe (the run_isolated function). The parent process blocks on a read() call until the child has completed its full measurement campaign and has serialized its BenchmarkResults structure back through the pipe, after which the child is reaped through waitpid() to avoid leaving zombie processes. This design ensures that the memory-allocation history, the heap fragmentation state, and the resident-set size accumulated while benchmarking McEliece, whose public key and, above all, private key are considerably larger than those of HQC, cannot bleed into, and therefore distort, the peak-memory measurement subsequently taken for HQC, since getrusage(RUSAGE_SELF, ...) reports the maximum resident set size accumulated over the entire lifetime of the calling process. Running the two algorithms in separate address spaces is therefore a safeguard to ensure a correct comparison of their memory footprints.

3.2.3 Timing and Cycle-Counting Instrumentation

Two independent and complementary metrics are collected for every timed operation. Wall-clock latency is measured in nanoseconds through clock_gettime(CLOCK_MONOTONIC_RAW, ...) implemented in the harness by get_time_ns, a clock source chosen specifically because it is immune to NTP-induced slewing and frequency adjustment, which makes it preferable to CLOCK_MONOTONIC for the purpose of measuring short intervals with sub-microsecond resolution. In parallel, the harness reads the processor's timestamp counter directly through the rdtsc instruction (via inline assembly, implemented by get_cpu_cycles, hence the restriction of the harness to the x86/x86-64 architecture family), producing a raw CPU-cycle count for each operation.
The purpose of collecting both metrics is that they provide complementary information. Wall-clock time is the most relevant measure for application developers and system integrators, as it reflects the actual execution time experienced in practice. In contrast, CPU cycle count is less affected by temporary frequency-scaling effects and therefore provides a more stable and hardware-independent estimate of the

intrinsic computational cost of an operation. Comparing these two metrics also helps identifying any remaining frequency-scaling artifacts by verifying whether the performance ratio between the two algorithms is consistent in both the time-based and cycle-based measurements.

3.2.4 Statistical Treatment of the Measurements

For each of the three timed operations (key generation, encapsulation, decapsulation), the harness executes NUM_ITERATIONS $= 10000$ repetitions and computes, over the resulting sample, the arithmetic mean and the sample standard deviation, implemented respectively by calculate_mean and calculate_stddev. Around each mean, a 95% confidence interval half-width is derived by calculate_confidence_interval, which multiplies the standard deviation, normalized by the square root of the sample size, by a Student's t critical value looked up for the corresponding sample size.
Furthermore, the harness reports, for every operation and every algorithm, the coefficient of variation, defined as the ratio of the standard deviation to the mean, expressed as a percentage. Each value is classified into one of three bands, "very stable," "stable," or "variable." Together with the confidence intervals, this gives the reader a measure of how much each operation's timing fluctuates from one execution to the next, and hence of how much weight a comparison of point estimates can bear.

3.2.5 Correctness Verification

Every benchmarking run also performs a functional correctness check, entirely orthogonal to the timing measurements: at the end of the encapsulation/decapsulation loop, the shared secret produced by the encapsulating party (ss_enc) is compared, via memcmp, against the shared secret recovered by the decapsulating party (ss_dec), and the outcome (SUCCESS or FAILED) is reported. This verification constitutes a minimal but necessary sanity check that the benchmarked build of the reference implementation is functionally correct on the target platform before its performance figures are taken at face value; a performance measurement obtained from a decapsulation routine that silently fails to recover the correct shared secret would be, at best, meaningless.

3.3 Description of the Test Suite and of the Code Structure

The benchmark harness is organized into four functional blocks, which are described below in the order in which they are exercised at run time, so as to make explicit the logical flow that produces the final report.

3.3.1 Helper and Infrastructure Functions

This block groups the low-level facilities on which the rest of the harness depends, implementing in code the methodological choices motivated in the preceding two subsections. The CPU-affinity and process-priority setters (set_cpu_affinity, set_high_priority, Listings 3.1–3.2) realize the environmental-noise controls described above: the former pins the benchmarking process to a single logical core through sched_setaffinity, first validating the requested core index against sysconf (_SC_NPROCESSORS_ONLN) to avoid an invalid affinity mask, while the latter attempts to raise the process's scheduling priority to its maximum value (-20) via setpriority. Both functions are deliberately written as best-effort: rather than aborting execution, they emit a warning to stderr and allow the harness to continue under default scheduling conditions if the underlying system call fails, which is expected when the process lacks the elevated privileges normally required to modify priority or affinity.

The peak-memory accessor (get_peak_memory_kb, Listing 3.3) wraps getrusage (RUSAGE_SELF, ...) to report the maximum resident set size accumulated by the calling process; as discussed above, its process-lifetime scope is precisely why each algorithm is benchmarked in an isolated child process, so that the value it returns reflects only the memory footprint of the algorithm currently under test.

The two timing primitives, get_time_ns and get_cpu_cycles (Listings 3.4–3.5), implement respectively the wall-clock and cycle-count channels introduced previously: the former queries CLOCK_MONOTONIC_RAW through clock_gettime and converts the result to a single double-precision nanosecond value, while the latter issues the rdtsc instruction directly via inline assembly, reconstructing the 64-bit cycle count from its high and low 32-bit halves returned in the edx and eax registers. Its declaration as static inline minimizes function-call overhead at the measurement

boundary, which matters at the sub-microsecond resolution these primitives are meant to achieve.

Beyond these, the block also includes the CPU and per-algorithm warm-up routines (warmup_cpu, warmup_algorithm_mceliece, warmup_algorithm_hqc), the statistical primitives (calculate_mean, calculate_stddev, calculate_confidence_interval), and a small hexadecimal pretty-printer (print_hex) used to render shared-secret values in the final report.

```
76 void set_cpu_affinity(int cpu_id) {
77     long nprocs = sysconf(_SC_NPROCESSORS_ONLN);
78     if (nprocs > 0 && (cpu_id < 0 || cpu_id >= nprocs)) {
79         fprintf(stderr, "Warning: requested CPU %d out of range
       ↪  (0..%ld). Skipping affinity.\n", cpu_id, nprocs - 1);
80         return;
81     }
82 
83     cpu_set_t cpuset;
84     CPU_ZERO(&cpuset);        // clear all CPUs in the set
85     CPU_SET(cpu_id, &cpuset); // add the requested CPU to the set
86     if (sched_setaffinity(0, sizeof(cpu_set_t), &cpuset) != 0) {
87         perror("Warning: sched_setaffinity failed");
88         printf("Continuing without CPU affinity...\n");
89     }
90 }
```

Listing 3.1 Function to set CPU affinity (best-effort)

```
93 void set_high_priority() {
94     if (setpriority(PRIO_PROCESS, 0, -20) != 0) {
95         perror("Warning: setpriority failed (requires root)");
96         printf("Continuing with normal priority...\n");
97     }
98 }
```

Listing 3.2 Function to set high priority

```
120 size_t get_peak_memory_kb() {
121     struct rusage usage;
122     if (getrusage(RUSAGE_SELF, &usage) == 0) {
123         return usage.ru_maxrss;  // KB in Linux
124     }
125     return 0;
126 }
```

Listing 3.3 Function to get peak memory in KB

```
113 double get_time_ns() {
114     struct timespec ts;
115     clock_gettime(CLOCK_MONOTONIC_RAW, &ts);
116     return ts.tv_sec * 1e9 + ts.tv_nsec;
117 }
```

Listing 3.4 Function to return nanoseconds

```
69 static inline uint64_t get_cpu_cycles() {
70     unsigned int lo, hi;
71     __asm__ __volatile__ ("rdtsc" : "=a" (lo), "=d" (hi));
72     return ((uint64_t)hi << 32) | lo;
73 }
```

Listing 3.5 Function to read the CPU timestamp counter (x86/x86_64 only)

3.3.2 Process-Isolation Layer

The run_isolated function (Listing 3.6) implements the fork/pipe/waitpid pattern motivated above providing the isolated address space required to keep the memory-allocation history of one algorithm from contaminating the peak-memory measurement of the next. It takes as arguments a function pointer bench_fn, matching the signature of the per-algorithm benchmarking routines (such as benchmark_mceliece), and the number of iterations to run, so that the same isolation logic can be reused unchanged across algorithms.

A unidirectional pipe is first created with pipe(pipefd), after which fork() splits execution into parent and child. In the child branch, the unused read end of the pipe is closed, the benchmark function is invoked directly, and its return value, a BenchmarkResults structure, the plain-old-data aggregate holding every mean, standard deviation, cycle count, size, and extremal-timing field described so far, is written back to the parent as a single raw, byte-for-byte block through write, with an explicit fflush(stdout) beforehand to ensure that no buffered progress output from the benchmark is left pending when the child terminates via. In the parent branch, the unused write end is closed instead, and a blocking read() on the pipe suspends the parent until the full structure has been received; the number of bytes actually read is checked against sizeof(results) so that a short read, for instance caused by the child crashing before completing its write, is reported as a warning rather than silently producing a partially initialized result. Finally, waitpid() reaps the child process, preventing it from lingering as a zombie, and the reconstructed BenchmarkResults is returned to the caller exactly as if the benchmark had been run in-process, while having in fact executed in complete memory isolation.

```
177 BenchmarkResults run_isolated(BenchmarkResults (*bench_fn)(int), int
    ↪ iterations) {
178 int pipefd[2];  // unidirectional pipe: pipefd[0] = read end,
    ↪ pipefd[1] = write end
179 if (pipe(pipefd) != 0) {
180     perror("pipe failed");
181     BenchmarkResults empty = {0};
182     return empty;
183 }
184
185 pid_t pid = fork();
186 if (pid < 0) {
187     perror("fork failed");
188     BenchmarkResults empty = {0};
189     return empty;
190 }
191
```

```
192     if (pid == 0) {
193         // Child: run the benchmark, send results back through the
            ↪ pipe
194         close(pipefd[0]);   // no need to rread
195         BenchmarkResults results = bench_fn(iterations);
196         fflush(stdout);     // make sure no print are pending
197         write(pipefd[1], &results, sizeof(results));    // send
            ↪ results to parent
198         close(pipefd[1]);
199         _exit(0);
200     }
201
202     // Parent: block until the child sends its results, then reap it
203     close(pipefd[1]);   // no need to write
204     BenchmarkResults results = {0};
205     ssize_t n = read(pipefd[0], &results, sizeof(results));
206     close(pipefd[0]);
207     if (n != (ssize_t)sizeof(results)) {
208         fprintf(stderr, "Warning: incomplete results from child\n");
209     }
210
211     int status;
212     waitpid(pid, &status, 0);   // no zombies
213     return results;
214 }
```

Listing 3.6 Function to run a benchmark in an isolated child process

3.3.3 Benchmark Functions

The core measurement logic of the benchmark is implemented by two functions, benchmark_mceliece and benchmark_hqc, each of type BenchmarkResults$(*)$(int), as they take the number of iterations as input and return the statistics produced. They are executed inside an isolated child process by run_isolated described above, so that the two algorithms are measured under identical, freshly initialized system conditions and cannot interfere with each other's memory footprint or CPU state. The two functions are structurally identical: the only difference lies in

which PQClean API is invoked (PQCLEAN_MCELIECE6960119_CLEAN_$*$ versus PQCLEAN_HQC256_CLEAN_$*$), so they are described together.

Each function begins by populating the BenchmarkResults struct with the fixed, algorithm-dependent sizes reported by the PQClean API (public key, secret key, ciphertext, and shared-secret size). The benchmarking environment is then prepared by pinning the process to a single CPU core, requesting the highest scheduling priority available, warming up the CPU with a dummy floating-point loop, and running a small number of "throwaway" key-generation calls (warmup_algorithm_mceliece / warmup_algorithm_hqc), (Listing 3.7). This sequence removes cold-cache and frequency-scaling effects from the very first measured iteration.

```
291 printf("Setting up benchmark environment...\n");
292 set_cpu_affinity(0);
293 set_high_priority();
294 warmup_cpu();
295 warmup_algorithm_mceliece();
```

Listing 3.7 Environment preparation before timing (McEliece variant)

After allocating the arrays that will hold, for every iteration, the wall-clock time (in milliseconds) and the raw CPU cycle count of each of the three KEM operations, the function enters its main measurement loop. On each iteration, key generation, encapsulation, and decapsulation are bracketed by a pair of timestamps: get_time_ns and get_cpu_cycles, in order to obtain the wall-clock time in milliseconds and the number of CPU cycles, respectively (Listing 3.8).

```
348 uint64_t kg_cstart = get_cpu_cycles();
349 double kg_start = get_time_ns();
350 PQCLEAN_MCELIECE6960119_CLEAN_crypto_kem_keypair(pk, sk);
351 double kg_end = get_time_ns();
352 uint64_t kg_cend = get_cpu_cycles();
353 keygen_times[i]  = (kg_end - kg_start) / 1e6;   // ns -> ms
354 keygen_cycles[i] = (double)(kg_cend - kg_cstart);
```

Listing 3.8 Timed key-generation step inside the benchmark loop

Encapsulation and decapsulation are timed with the same pattern.
Finally, each iteration performs a correctness check by comparing the shared secret produced by encapsulation against the one recovered by decapsulation via memcmp (Listing 3.9); any mismatch increments a failures counter, allowing the harness to flag a broken build even if the timing results themselves look plausible.

```
394 if (memcmp(ss_enc, ss_dec, PQCLEAN_MCELIECE6960119_CLEAN_CRYPTO_BYTES)
    ↪  != 0) {
395     failures++;
396 }
```

Listing 3.9 Correctness check after each iteration

Once all iterations have completed, the function derives the mean and standard deviation of both the wall-clock times and the cycle counts for each of the three operations using calculate_mean and calculate_stddev, and records the process's peak resident memory via get_peak_memory_kb. It prints a pass/fail summary of the correctness check, releases all dynamically allocated buffers, and returns the populated BenchmarkResults struct to run_isolated, which relays it to the parent process over the pipe. The resulting struct is later consumed by print_results and print_comparison to produce the per-algorithm reports and the head-to-head statistical comparison discussed below.

3.3.4 Reporting Functions

The reporting functions are divided in two categories. First print_results renders, for a single algorithm, a structured report comprising key/ciphertext sizes, the mean and standard deviation of each timed operation, the mean cycle counts, the coefficient of variation with its qualitative stability label, the peak memory usage, the minimum/-maximum encapsulation and decapsulation times together with the associated shared secrets, and the aggregate end-to-end latency.
Last print_comparison consumes the BenchmarkResults of both algorithms and produces a head-to-head report: relative size ratios for public key, private key, and ciphertext; the memory-usage differential; per-operation speed ratios accompanied by 95%-confidence-interval figures; aggregate total-time and total-cycle-count comparisons; a stability comparison based on the average coefficient of variation

across the three operations; and a closing qualitative summary that flags, for each of the dimensions considered (key size, speed, memory, stability), which of the two schemes is preferable, while explicitly reaffirming that the two parameter sets are cryptographically equivalent in terms of the security level they target.

The main routine orchestrates the whole campaign: it prints a configuration banner reporting the number of iterations, the number of warm-up iterations, the clock source in use, and the environmental controls that have been enabled, before invoking run_isolated on the fixed-key benchmark of each algorithm and feeding the resulting structures to print_results and print_comparison in sequence.

3.4 Results

This section presents the empirical results obtained from the benchmark described above, organized by NIST security level. For each level, the relevant Classic McEliece parameter sets are compared against the corresponding HQC parameter set along five dimensions: key and ciphertext size, wall-clock timing, CPU-cycle counts, measurement variability, and peak memory usage, each summarized in a dedicated table together with an indication of the better-performing scheme for every metric. A closing table for each level then distills these results into a qualitative summary, highlighting the trade-offs between the two schemes at equivalent security strength.
Level 5 additionally includes a dedicated comparison among the three Classic McEliece parameter sets available at that level, isolating the effect of code length and error-correction weight on performance independently of the McEliece–HQC comparison.

3.4.1 NIST Security Level 1

mceliece348864 **vs** mceliece348864f **vs** hqc-128

Object	**ME-348864 (bytes)**	**ME-348864f (bytes)**	**HQC-128 (bytes)**	**Smaller**
Public key	261120	261120	2249	HQC-128
Private key	6492	6492	2305	HQC-128
Ciphertext	96	96	4433	ME-348864 and ME-348864f
Total size	267708	267708	8987	HQC-128

Table 3.2 Size comparison between McEliece348864, McEliece348864f and HQC-128

Operation	**ME-348864 (ms)**	**ME-348864f (ms)**	**HQC-128 (ms)**	**Faster**
Key generation	93.505 ± 1.035	48.790 ± 0.018	1.127 ± 0.001	HQC-128
Encapsulation	0.037 ± 0.000	0.036 ± 0.000	2.279 ± 0.002	ME-348864f
Decapsulation	23.235 ± 0.008	23.212 ± 0.008	3.476 ± 0.003	HQC-128
Total time	116.776 ± 1.035	72.038 ± 0.020	6.882 ± 0.004	HQC-128

Table 3.3 Speed comparison between McEliece348864, McEliece348864f and HQC-128 (mean $\pm$ 95% confidence interval, $n = 10000$)

Operation	**ME-348864 (cycles)**	**ME-348864f (cycles)**	**HQC-128 (cycles)**	**Faster**
Key generation	214556470 ± 2374718	111954359 ± 41510	2586038 ± 2722	HQC-128
Encapsulation	84690 ± 359	82436 ± 345	5229774 ± 5026	ME-348864f
Decapsulation	53314959 ± 17385	53263405 ± 17592	7976486 ± 6303	HQC-128
Total cycles	267956120 ± 2374782	165300201 ± 45085	15792298 ± 8509	HQC-128

Table 3.4 CPU-cycle comparison between McEliece348864, McEliece348864f and HQC-128 (mean $\pm$ 95% confidence interval, $n = 10000$)

Operation	ME-348864 (value)	ME-348864f (value)	HQC-128 (value)
Key generation	55.79% (variable)	1.87% (very stable)	5.31% (stable)
Encapsulation	21.39% (variable)	21.16% (variable)	4.84% (very stable)
Decapsulation	1.64% (very stable)	1.66% (very stable)	3.98% (very stable)

Table 3.5 Variability for McEliece348864, McEliece348864f and HQC-128

Metric	ME-348864	ME-348864f	HQC-128
Peak memory usage	2212 KB	2324 KB	1664 KB
Average coefficient of variation	26.27%	8.23%	4.71%

Table 3.6 Memory and stability comparison between McEliece348864, McEliece348864f and HQC-128

Criterion	Winner
Smaller keys (pk + sk)	HQC-128
Faster overall (total time)	HQC-128
Lower peak memory	HQC-128
More stable (lower CV)	HQC-128
Security level	Equivalent (all Level 1)

Table 3.7 Summary of the McEliece348864 vs. McEliece348864f vs. HQC-128

3.4.2 NIST Security Level 3

mceliece460896 **vs** mceliece460896f **vs** hqc-192

Object	**ME-460896 (bytes)**	**ME-460896f (bytes)**	**HQC-192 (bytes)**	**Smaller**
Public key	524160	524160	4522	HQC-192
Private key	13608	13608	4586	HQC-192
Ciphertext	156	156	8978	ME-460896 and ME-460896f
Total size	537924	537924	18086	HQC-192

Table 3.8 Size comparison between McEliece460896, McEliece460896f and HQC-192

Operation	**ME-460896 (ms)**	**ME-460896f (ms)**	**HQC-192 (ms)**	**Faster**
Key generation	252.251 ± 2.637	141.585 ± 0.054	3.451 ± 0.002	HQC-192
Encapsulation	0.097 ± 0.001	0.093 ± 0.001	6.935 ± 0.004	ME-460896f
Decapsulation	46.921 ± 0.015	46.991 ± 0.018	10.443 ± 0.007	HQC-192
Total time	299.269 ± 2.637	188.669 ± 0.057	20.829 ± 0.008	HQC-192

Table 3.9 Speed comparison between McEliece460896, McEliece460896f and HQC-192 (mean $\pm$ 95% confidence interval, $n = 10000$)

Operation	**ME-460896 (cycles)**	**ME-460896f (cycles)**	**HQC-192 (cycles)**	**Faster**
Key generation	578818015 ± 6051594	324883233 ± 124630	7918147 ± 5601	HQC-192
Encapsulation	222722 ± 1253	212935 ± 1240	15913181 ± 10059	ME-460896f
Decapsulation	107666031 ± 35402	42308	23962736 ± 15113	HQC-192
Total cycles	686706768 ± 6051698	432923171 ± 131622	47794064 ± 18999	HQC-192

Table 3.10 CPU-cycle comparison between McEliece460896, McEliece460896f and HQC-192 (mean $\pm$ 95% confidence interval, $n = 10000$)

Operation	ME-460896 (value)	ME-460896f (value)	HQC-192 (value)
Key generation	52.70% (variable)	1.93% (very stable)	3.57% (very stable)
Encapsulation	28.42% (variable)	29.41% (variable)	3.19% (very stable)
Decapsulation	1.66% (very stable)	1.98% (very stable)	3.18% (very stable)

Table 3.11 Variability for McEliece460896, McEliece460896f and HQC-192

Metric	ME-460896	ME-460896f	HQC-192
Peak memory usage	2924 KB	2924 KB	1664 KB
Average coefficient of variation	27.59%	11.11%	3.31%

Table 3.12 Memory and stability comparison between McEliece460896, McEliece460896f and HQC-192

Criterion	Winner
Smaller keys (pk + sk)	HQC-192
Faster overall (total time)	HQC-192
Lower peak memory	HQC-192
More stable (lower CV)	HQC-192
Security level	Equivalent (all Level 3)

Table 3.13 Summary of the McEliece460896 vs. McEliece460896f vs. HQC-192

3.4.3 NIST Security Level 5

mceliece6688128f **vs** hqc-256

Object	**ME-6688128f (bytes)**	**HQC-256 (bytes)**	**Smaller**
Public key	1044992	7245	HQC-256
Private key	13932	7317	HQC-256
Ciphertext	208	14421	ME-6688128f
Total size	1059132	28983	HQC-256

Table 3.14 Size comparison between McEliece6688128f and HQC-256

Operation	**ME-6688128f (ms)**	**HQC-256 (ms)**	**Faster**
Key generation	249.159 ± 0.087	6.195 ± 0.003	HQC-256
Encapsulation	0.185 ± 0.001	12.482 ± 0.006	ME-6688128f
Decapsulation	89.029 ± 0.027	18.839 ± 0.008	HQC-256
Total time	338.374 ± 0.091	37.516 ± 0.011	HQC-256

Table 3.15 Speed comparison between McEliece6688128f and HQC-256 (mean ± 95% confidence interval, $n = 10000$)

Operation	**ME-6688128f (cycles)**	**HQC-256 (cycles)**	**Faster**
Key generation	571722184 ± 198617	14214942 ± 7573	HQC-256
Encapsulation	426055 ± 2333	28641421 ± 14429	ME-6688128f
Decapsulation	204287422 ± 60955	43228937 ± 18361	HQC-256
Total cycles	776435661 ± 207773	86085300 ± 24550	HQC-256

Table 3.16 CPU-cycle comparison between McEliece6688128f and HQC-256 (mean ± 95% confidence interval, $n = 10000$)

Operation	**ME-6688128f (value)**	**HQC-256 (value)**
Key generation	1.75% (very stable)	2.69% (very stable)
Encapsulation	27.63% (variable)	2.54% (very stable)
Decapsulation	1.50% (very stable)	2.14% (very stable)

Table 3.17 Variability for McEliece6688128f and HQC-256

Metric	**ME-6688128f**	**HQC-256**
Peak memory usage	4100 KB	1792 KB
Average coefficient of variation	10.29%	2.46%

Table 3.18 Memory and stability comparison between McEliece6688128f and HQC-256

Criterion	**Winner**
Smaller keys (pk + sk)	HQC-256
Faster overall (total time)	HQC-256
Lower peak memory	HQC-256
More stable (lower CV)	HQC-256
Security level	Equivalent (both Level 5)

Table 3.19 Summary of the McEliece6688128f vs. HQC-256 comparison

mceliece6960119f **vs** hqc-256

Object	**ME-6960119f (bytes)**	**HQC-256 (bytes)**	**Smaller**
Public key	1047319	7245	HQC-256
Private key	13948	7317	HQC-256
Ciphertext	194	14421	ME-6960119f
Total size	1061461	28983	HQC-256

Table 3.20 Size comparison between McEliece6960119f and HQC-256

Operation	**ME-6960119f (ms)**	**HQC-256 (ms)**	**Faster**
Key generation	212.011 ± 0.068	6.183 ± 0.004	HQC-256
Encapsulation	0.241 ± 0.001	12.454 ± 0.007	ME-6960119f
Decapsulation	86.225 ± 0.027	18.930 ± 0.009	HQC-256
Total time	298.476 ± 0.073	37.567 ± 0.012	HQC-256

Table 3.21 Speed comparison between McEliece6960119f and HQC-256 (mean ± 95% confidence interval, $n = 10000$)

Operation	**ME-6960119f (cycles)**	**HQC-256 (cycles)**	**Faster**
Key generation	486481275 ± 156366	14187194 ± 8126	HQC-256
Encapsulation	553935 ± 1949	28577973 ± 16309	ME-6960119f
Decapsulation	197852205 ± 60972	43436516 ± 20783	HQC-256
Total cycles	684887415 ± 167844	86201683 ± 27640	HQC-256

Table 3.22 CPU-cycle comparison between McEliece6960119f and HQC-256 (mean ± 95% confidence interval, $n = 10000$)

Operation	**ME-6688128f (value)**	**HQC-256 (value)**
Key generation	1.62% (very stable)	2.89% (very stable)
Encapsulation	17.74% (variable)	2.88% (very stable)
Decapsulation	1.55% (very stable)	2.41% (very stable)

Table 3.23 Variability for McEliece6960119f and HQC-256

Metric	**ME-6960119f**	**HQC-256**
Peak memory usage	4056 KB	1792 KB
Average coefficient of variation	6.97%	2.72%

Table 3.24 Memory and stability comparison between McEliece6960119f and HQC-256

Criterion	Winner
Smaller keys (pk + sk)	HQC-256
Faster overall (total time)	HQC-256
Lower peak memory	HQC-256
More stable (lower CV)	HQC-256
Security level	Equivalent (both Level 5)

Table 3.25 Summary of the McEliece6960119f vs. HQC-256 comparison

mceliece8192128f **vs** hqc-256

Object	ME-8192128f (bytes)	HQC-256 (bytes)	Smaller
Public key	1357824	7245	HQC-256
Private key	14120	7317	HQC-256
Ciphertext	208	14421	ME-8192128f
Total size	1372152	28983	HQC-256

Table 3.26 Size comparison between McEliece8192128f and HQC-256

Operation	ME-8192128f (ms)	HQC-256 (ms)	Faster
Key generation	279.856 ± 0.091	6.293 ± 0.004	HQC-256
Encapsulation	0.216 ± 0.004	12.663 ± 0.008	ME-8192128f
Decapsulation	109.679 ± 0.030	19.109 ± 0.010	HQC-256
Total time	389.751 ± 0.096	38.065 ± 0.013	HQC-256

Table 3.27 Speed comparison between McEliece8192128f and HQC-256 (mean ± 95% confidence interval, $n = 10000$)

Operation	**ME-8192128f (cycles)**	**HQC-256 (cycles)**	**Faster**
Key generation	642158944 ± 208346	14439119 ± 9726	HQC-256
Encapsulation	496213 ± 2438	29057067 ± 17302	ME-8192128f
Decapsulation	251670667 ± 68265	43848892 ± 23459	HQC-256
Total cycles	894325824 ± 219258	87345078 ± 30729	HQC-256

Table 3.28 CPU-cycle comparison between McEliece8192128f and HQC-256 (mean ± 95% confidence interval, $n = 10000$)

Operation	**ME-6688128f (value)**	**HQC-256 (value)**
Key generation	1.64% (very stable)	3.40% (very stable)
Encapsulation	29.79% (variable)	3.00% (very stable)
Decapsulation	1.37% (very stable)	2.70% (very stable)

Table 3.29 Variability for McEliece8192128f and HQC-256

Metric	**ME-8192128f**	**HQC-256**
Peak memory usage	4796 KB	1792 KB
Average coefficient of variation	10.93%	3.03%

Table 3.30 Memory and stability comparison between McEliece8192128f and HQC-256

Criterion	**Winner**
Smaller keys (pk + sk)	HQC-256
Faster overall (total time)	HQC-256
Lower peak memory	HQC-256
More stable (lower CV)	HQC-256
Security level	Equivalent (both Level 5)

Table 3.31 Summary of the McEliece8192128f vs. HQC-256 comparison

mceliece6688128 **vs** mceliece6960119 **vs** mceliece8192128

Object	**ME-6688128 (bytes)**	**ME-6960119 (bytes)**	**ME-8192128 (bytes)**	**Smaller**
Public key	1044992	1047319	1357824	ME-6688128
Private key	13932	13948	14120	ME-6688128
Ciphertext	208	194	208	ME-6960119
Total size	1059132	1061461	1372152	ME-6688128

Table 3.32 Size comparison between McEliece6688128, McEliece6960119 and McEliece8192128

Operation	**ME-6688128 (ms)**	**ME-6960119 (ms)**	**ME-8192128 (ms)**	**Faster**
Key generation	614.736 ± 8.864	554.217 ± 8.043	705.041 ± 10.534	ME-6960119
Encapsulation	0.212 ± 0.001	0.249 ± 0.001	0.225 ± 0.001	ME-6688128
Decapsulation	89.632 ± 0.032	86.828 ± 0.027	109.970 ± 0.033	ME-6960119
Total time	704.580 ± 8.864	641.294 ± 8.043	815.236 ± 10.534	ME-6960119

Table 3.33 Speed comparison between McEliece6688128, McEliece6960119 and McEliece8192128 (mean ± 95% confidence interval, $n = 10000$)

Operation	**ME-6688128 (cycles)**	**ME-6960119 (cycles)**	**ME-8192128 (cycles)**	**Faster**
Key generation	1410579032 ± 20339306	1271710806 ± 18456432	1617793281 ± 24170527	ME-6960119
Encapsulation	486651 ± 2306	571325 ± 1839	517781 ± 2332	ME-6688128
Decapsulation	205670159 ± 72963	199237473 ± 62495	252337285 ± 76231	ME-6960119
Total cycles	1616735842 ± 20339437	1471519604 ± 18456538	1870648347 ± 24170647	ME-6960119

Table 3.34 CPU-cycle comparison between McEliece6688128, McEliece6960119 and McEliece8192128 (mean ± 95% confidence interval, $n = 10000$)

Operation	ME-6688128 (value)	ME-6960119 (value)	ME-8192128 (value)
Key generation	72.68% (variable)	73.15% (variable)	75.30% (variable)
Encapsulation	23.90% (variable)	16.23% (variable)	22.72% (variable)
Decapsulation	1.79% (very stable)	1.58% (very stable)	1.52% (very stable)

Table 3.35 Variability for McEliece6688128, McEliece6960119 and McEliece8192128

Metric	ME-6688128	ME-6960119	ME-8192128
Peak memory usage	4100 KB	4184 KB	4924 KB
Average coefficient of variation	32.79%	30.32%	33.18%

Table 3.36 Memory and stability comparison between McEliece6688128, McEliece6960119 and McEliece8192128

Criterion	Winner
Smaller keys (pk + sk)	ME-6688128
Faster overall (total time)	ME-6960119
Lower peak memory	ME-6688128
More stable (lower CV)	ME-6960119
Security level	Equivalent (all Level 5)

Table 3.37 Summary of the McEliece6688128 vs. McEliece6960119 vs. McEliece8192128

3.5 Conclusions

3.5.1 McEliece vs HQC at Each Level

The absolute gap in total time between McEliece and HQC grows steadily as security increases: about 110 ms at Level 1, 278 ms at Level 3, and 260-350 ms at Level 5 depending on the variant. This is expected, since key generation, which is McEliece's main bottleneck, involves Gaussian elimination on a matrix whose size grows with

the code length n, so its cost grows faster than HQC's, whose key generation only requires sampling a few random polynomials. Interestingly, the relative ratio between McEliece and HQC does not grow in the same simple way: it is about 17× at Level 1, drops to about 14× at Level 3, and drops further to roughly 8-10× at Level 5 for the "f" variants. The reason is that the Level 5 comparison uses only "f" variants, whose optimized key generation keeps the keygen ratio against HQC fairly stable across levels (roughly 34× to 48×), while the standard (non-"f") variant used at Levels 1 and 3 has a much worse, and less consistent, keygen ratio (73× to 83×). In short, McEliece's disadvantage in absolute time keeps growing with security level, but switching to the "f" variant partially compensates for this as the parameters get larger.

The same qualitative pattern repeats at every level: HQC wins on key size, ciphertext-plus-key total size, total time, memory, and stability, while McEliece wins on encapsulation speed and ciphertext size. This is a direct consequence of how the two schemes work. McEliece's encapsulation is just one matrix-vector multiplication with a fixed-weight error vector, so it is extremely cheap and has almost no dependency on t (the number of correctable errors); its decapsulation, however, requires running a full syndrome-decoding algorithm to find and correct t errors in a code of length n, which is by far the most expensive step besides key generation. HQC works the other way around: both encapsulation and decapsulation involve multiplying and decoding over a structured polynomial ring, which is more balanced but consistently more expensive than McEliece's, hence why HQC is always slower at encapsulation despite being faster everywhere else.

3.5.2 McEliece variants at Level 5

Among mceliece6688128 ($n = 6688$, $t = 128$), mceliece6960119 ($n = 6960$, $t = 119$), and mceliece8192128 ($n = 8192$, $t = 128$), mceliece6960119 gives the best overall performance despite not having the smallest code length. This shows that performance is driven more by the number of errors t than by the code length n alone: a lower t (119 vs 128) means the decoding step, and to some extent the retry logic during key generation, has less work to do, more than compensating for a code length between the other two. mceliece8192128 pays the price of the largest n on every operation (largest keys, slowest keygen, slowest decapsulation) without gaining a real advantage in return, since its t is the same as mceliece6688128's. This

suggests that, when choosing among McEliece parameter sets at the same security level, the error-correction weight t is at least as important a design lever as the raw code length n.

3.5.3 McEliece Standard vs “f” Variants

The “f” variants show what the retry mechanism costs in the standard variants. During key generation, McEliece must reduce part of its parity-check matrix to systematic form via Gaussian elimination; if this reduction fails, which happens with non-negligible probability, the whole key generation must restart with a new random seed. The standard variant requires strict systematic form and therefore restarts often, which explains both why its mean key-generation time is high and why its coefficient of variation is so large (52-73%): some keys are generated on the first attempt, others only after several retries, so the sample is a mixture of very different execution paths. The “f” variant instead accepts the more permissive semi-systematic form, which succeeds almost immediately, removing most restarts. This is why the “f” variants cut mean key-generation time roughly in half at every level and simultaneously collapse the coefficient of variation from "variable" (>20%) to "very stable" (around 2%): the improvement in stability is not a side effect, but the same underlying cause (fewer retries) as the improvement in speed. Encapsulation and decapsulation are, as expected, essentially unchanged between the two variants, since the semi-systematic relaxation only touches key generation.

3.5.4 Conclusions

The benchmark shows that Classic McEliece and HQC address fundamentally different points in the design space rather than being interchangeable at equal security levels. McEliece pays almost all of its cost upfront, in key generation and, to a lesser extent, decapsulation, in exchange for extremely fast and small-ciphertext encapsulation, while HQC spreads a moderate, well-balanced cost across all three operations and keeps its keys and ciphertexts far smaller overall. This makes HQC the more sensible default for most interactive or bandwidth-constrained scenarios, while McEliece becomes attractive specifically in scenarios where a key pair is generated once and reused for many encapsulations, and where the number of ciphertexts sent matters more than the one-time cost of generating the key. Within the

McEliece family itself, the results argue against simply picking the parameter set with the largest code length: mceliece6960119 outperforms mceliece8192128 across almost every metric thanks to its lower t, and the "f" variants should be preferred over the standard ones in essentially all cases, since they offer a large, consistent improvement in both speed and stability with no measurable downside.

Chapter 4

Analysis of Side Channel Attacks on McEliece and HQC Key Encapsulation Mechanism

4.1 Introduction to Side-Channel Attacks

The security proofs discussed in Chapter 1 and the concrete instantiations described in Chapter 2 are based on an adversary model: the adversary is only allowed to interact with the public key pk, the ciphertext c, and, in the IND-CCA setting, a decapsulation oracle $\mathsf{Decaps}_{\mathsf{sk}}(\cdot)$ that returns either a session key or a failure symbol $\perp$. This model is the correct one for reasoning about the algorithmic hardness of the underlying problems (the syndrome decoding problem, the linear code equivalence problem), but it does not capture what happens when a scheme is executed on physical hardware.

A side-channel attack (SCA) exploits information that leaks from a concrete implementation of a cryptographic primitive during its execution, like timing, power consumption, electromagnetic (EM) emanation, cache behaviour, or sound, rather than attacking the mathematical problem itself. Side-channel attacks were formalized by Kocher in 1996 [15] and have since been used successfully against essentially every class of cryptographic scheme, including code-based post-quantum candidates.

4.1.1 Leakage Detection with TVLA and the Welch t-test

To evaluate whether an implementation leaks sensitive information, evaluators commonly use the Test Vector Leakage Assessment (TVLA) framework. TVLA provides a standardised methodology to determine whether a device exhibits observable leakage without performing a full key-recovery attack. At the heart of TVLA is the Welch t-test, a statistical tool that compares the means of two leakage distributions obtained under different input conditions (e.g., fixed vs. random secrets, or two different key values).

Unlike the standard Student's t-test, the Welch version does not assume equal variances between the two populations. This makes it particularly suitable for side-channel measurements, where noise characteristics may differ significantly between traces. For each point in time t, the test computes a t-value defined as

$$t = \frac{\bar{X}_1 - \bar{X}_2}{\sqrt{\frac{s_1^2}{N_1} + \frac{s_2^2}{N_2}}},$$

where $\bar{X}_1$ and $\bar{X}_2$ are the sample means of the two groups, s_1^2 and s_2^2 are the sample variances, and N_1 and N_2 are the sample sizes.

In practice, a t-value exceeding a fixed threshold, typically ± 4.5, indicates that the difference between the two distributions is statistically significant, and thus that the device leaks information about the secret. This approach is non-specific: it does not require a particular leakage model, only the ability to separate measurements into two classes based on a known input or output value. As such, the Welch t-test has become the de facto standard for initial leakage detection in side-channel security evaluations.

4.1.2 Case Studies

This chapter presents two case studies illustrating how the gap between algorithmic and implementation security manifests in code-based cryptography, one per KEM finalist considered in this thesis.
The first section analyzes a key-recovery side-channel attack against the Classic McEliece KEM, based on the work of Guo, A. Johansson and T. Johansson [16], which exploits a leakage of the FFT by decoding specific ciphertexts containing

single errors.
The second section then turns to a chosen-ciphertext side-channel key-recovery attack against the HQC KEM, based on the work of Goy, Loiseau and Gaborit [17]. The attack targets the concrete decoding procedure used by the RMRS implementation of HQC, which is based on a concatenation of a Reed–Muller and a Reed–Solomon code.

4.2 A Key-Recovery Side-Channel Attack on Classic McEliece

For Classic McEliece, $\mathsf{Decaps}_{\mathsf{sk}}(\cdot)$ is based on the syndrome decoding of a binary Goppa code, and this section reviews the work of Guo, A. Johansson and T. Johansson [16], who show that this concrete decoding procedure (not the algebraic hardness argument of Chapter 2) leaks enough information through power consumption to recover sk in its entirety. Prior side-channel results against the scheme had only achieved message recovery; key recovery is more damaging, since a single successful attack compromises every ciphertext ever encrypted, past or future, under the exposed pk, rather than a single ciphertext. The attack is demonstrated against both an FPGA hardware implementation and a software implementation running on an ARM Cortex-M4 microcontroller.

4.2.1 Introduction

Recall Section 2.2 that the private code of McEliece is described by $\Gamma = (g, \alpha)$, where $g(x) \in \mathbb{F}_q[x]$, with $q = 2^m$, is an irreducible Goppa polynomial of degree t, and $\alpha = (\alpha_0, \ldots, \alpha_{n-1})$ is the support of n distinct elements of $\mathbb{F}_q$. The public key is the matrix T obtained by reducing the binary $mt \times n$ parity-check matrix of Γ to systematic form $[I_{mt} \,|\, T]$, i.e. $\mathsf{pk} = T$, and the secret key is $\mathsf{sk} = (\delta, g, \alpha, \mathbf{s})$, where $\mathbf{s} \in \mathbb{F}_2^n$ is the auxiliary string used for implicit rejection. $\mathsf{Encaps}_{\mathsf{pk}}$ samples a weight-t error vector $\mathbf{e} \in \mathbb{F}_2^n$ and outputs the syndrome $\mathbf{c} = [I_{mt} \,|\, T]\mathbf{e}^\top$, and $\mathsf{Decaps}_{\mathsf{sk}}(\mathbf{c})$ recovers $\mathbf{e}$ from $\mathbf{c}$ by syndrome decoding of Γ. Table 2.5 recalls the five official parameter sets, indexed by the degree m, the error-correction capability t, the code length n, and the resulting dimension $k = n - mt$.

The formal $\mathsf{Decaps}_{\mathsf{sk}}$ algorithm recalled above hides an implementation detail that is the actual target of the attack. In practice, recovering $\mathbf{e}$ from $\mathbf{c}$ is carried out with a doubled $2t \times n$ parity-check matrix together with a constant-time Berlekamp–Massey (BM) algorithm. The BM step outputs an error-locator polynomial $\Lambda(x) \in \mathbb{F}_q[x]$, playing the same role as the polynomial introduced in Chapter 1, but constructed here with roots directly at the error positions:

$$\Lambda(x) = \prod_{i \in J} (x - \alpha_i), \qquad J = \{\, i : e_i \neq 0 \,\}.$$

Evaluating $\Lambda(x)$ at every support point with an additive FFT then reconstructs $\mathbf{e}$: bit e_i is set to 1 exactly when $\Lambda(\alpha_i) = 0$.

The Core Vulnerability

The central observation exploited by the attack is deceptively simple. Once $\Lambda(x)$ has been computed by the Berlekamp–Massey step, the FFT evaluation of $\Lambda(x)$ over the full field $\mathbb{F}_q$ is a deterministic function of $\Lambda(x)$ alone.

This matters because the additive FFT is executed in constant time with respect to the control flow of the algorithm (a deliberate design choice to prevent classical timing attacks), but its data flow still depends entirely on the secret-influenced value $\Lambda(x)$. Two different values of $\Lambda(x)$ lead the FFT's steps through two different sequences of intermediate constant polynomials, and this variation is visible as a difference in the device's instantaneous power consumption.

The attack turns this into a key-recovery primitive by choosing ciphertexts corresponding to error vectors $\mathbf{e}_i$ of Hamming weight one, i.e., $J = \{i\}$ for a single chosen position i. For such a ciphertext, the recovered error-locator polynomial is forced to have the simple monic form

$$\Lambda(x) = x - \alpha_i,$$

so there are only $q = 2^m$ possible values of $\Lambda(x)$ (one per field element). Determining which of these q possibilities occurred during a given call to $\mathsf{Decaps}_{\mathsf{sk}}$ directly reveals α_i, i.e., one entry of the secret support α.

4.2.2 The Attack Algorithm

The attack follows the profiling/attack-phase: the adversary submits chosen ciphertexts to $\mathsf{Decaps}_{\mathsf{sk}}(\cdot)$, measures power consumption, and trains a classifier on a device under their own control without ever needing to know or clone sk. What is specific to Classic McEliece is the target of the classifier and the algebraic post-processing used to turn partial leakage into a full key, detailed below.

Profiling Phase

At the beginning the adversary samples a secret support α^{pub}.Then he computes, for a chosen index i, $\mathsf{Encaps}_{\mathsf{pk}}$ on the weight-one error vector $\mathbf{e}_i$ and submits the resulting ciphertexts to $\mathsf{Decaps}_{\mathsf{sk}}(\cdot)$, capturing the power traces of the additive FFT computation. This will produce an error locator polynomial $\Lambda(x) = x - \alpha_i^{pub}$. By repeating the process, the attacker obtains $q = 2^m$ different $\Lambda(x)$, and is now able to categorize all the traces according to the resulting polynomials.
The profiling phase is concluded by training a multilayer perceptron (MLP) on these labeled traces to predict α_i from an unseen trace.

Attack Phase

In the attack phase, this classifier is applied to traces captured from the actual target device. Two decision rules are considered:

1. **Naive approach**: accept the α_i with maximized likelihood;
2. **Threshold approach**: repeat the query for the same i and multiply the resulting (normalized) likelihood vectors component-wise until the maximum likelihood exceeds a chosen threshold τ. This trades additional oracle queries for higher reliability on the noisier, low-confidence entries, and is particularly useful on the more challenging software target.

Repeating this procedure for a chosen index set $\mathscr{I}$ recovers the corresponding support entries $\{\alpha_i : i \in \mathscr{I}\}$.

Recover $g(x)$

Recovering every α_i by brute force would already give full key recovery, but would require up to n oracle queries per key. The paper's main algebraic contribution is a much cheaper route: it suffices to recover the α_i values at the support positions of a single low-weight codeword of the public code.

Given a valid codeword $\mathbf{c} = (c_1, \ldots, c_n)$ with support $\mathscr{I}(\mathbf{c}) = \{i : c_i \neq 0\}$, the defining relation of the Goppa code, recalled in Chapter 1, implies that the auxiliary polynomial

$$c(x) \;=\; \sum_{i=1}^{n} \frac{c_i}{x-\alpha_i} \prod_{j \in I(\mathbf{c})} (x-\alpha_j)$$

is divisible by $g(x)$ in $\mathbb{F}_q[x]$, since

$$\sum_{i=1}^{n} \frac{c_i}{x-\alpha_i} = 0 \mod g(x).$$

Then, computing $c(x)$ only requires the α_i values at positions where $c_i \neq 0$, exactly the positions leaked via the side channel for that codeword. Factoring $c(x)$ into irreducibles (e.g., via Cantor–Zassenhaus) and selecting the irreducible factor of weight t then recovers the secret Goppa polynomial $g(x)$ directly, at a computational cost the authors show to be negligible (on the order of seconds, even for the largest parameter set).

This reduces the sample complexity of partial key recovery from n to $wt_H(\mathbf{c})$, the Hamming weight of the chosen codeword. Finding a low-weight codeword is a well-studied problem, information set decoding (ISD), which is resolved by implementing Stern's algorithm, giving a direct time/query trade-off: more computation invested in ISD yields a lower-weight codeword and hence fewer required oracle traces.

Recover the Full Key

Once $g(x)$ is known, recovering the remaining support entries depends on whether $n = 2^m$ (only the `mceliece8192128` parameter set) or $n < 2^m$ (the other four sets).

- **Case $n = 2^m$.** The support is then simply the entire field $\mathbb{F}_q$ in some secret order, and Sendrier's support splitting algorithm recovers the missing permutation

from $g(x)$ and pk alone, in $O(n^3)$ time and with no further oracle queries. Partial and full key recovery thus coincide in this case;

- **Case $n < 2^m$.** The support α is now an unknown proper subset of $\mathbb{F}_q$, which the support splitting algorithm cannot directly resolve. In this case the attack proceeds in three main steps:

  1. **Recover part of the matrix.** The attacker first recovers the first few columns of the secret parity-check matrix. This is possible because these columns correspond to a small, known portion of the support;
  2. **Build a lookup table.** For every possible field element α, the attacker precomputes the corresponding column that would appear in the parity-check matrix. This creates a table that maps each field element to its unique matrix column;
  3. **Recover the full support.** Instead of looking for a very low-weight codeword, the attacker looks for a codeword whose support mostly overlaps with the already known positions. Using side-channel information, the attacker recovers a few more support elements, then factorizes a polynomial to obtain the secret irreducible polynomial $g(x)$. Finally, the full support is recovered by matching each column of the parity-check matrix against the precomputed table.

  The table lookup makes the recovery efficient: once the matrix is known, each column reveals which field element it corresponds to. The attack only needs to guess a small number of support elements initially, and the rest are determined automatically.

4.2.3 Experimental Results

The attack was implemented end-to-end and evaluated with the ChipWhisperer platform against two real targets, a hardware and a software implementation of Classic McEliece, both instantiating the smallest parameter set, mceliece348864.

Hardware Implementation: FPGA

The hardware implementation targets a CW305 board (Xilinx Artix-7). The fixed-vs-random TVLA on the additive FFT segment of the trace shows a clear, statistically significant leak (test statistic exceeding the standard $|t| \geq 4.5$ threshold). A profiling set of $= 418{,}560$ traces across 30 random keys was used to train the MLP classifier. The model is then evaluated against 30 held-out keys not used in profiling (over 100,000 attack traces total), and the experiment achieved a 100% empirical success rate at recovering each targeted support entry from a single trace.

Software Implementation: ARM Cortex-M4

The software implementation targets an STM32F415RGT6 microcontroller running a CHES 2021 Cortex-M4 implementation. The much longer execution time (roughly 340 times more clock cycles than the FPGA) required down-sampling to capture the FFT step within the ChipWhisperer capture buffer. Despite this coarser sampling, piece-wise TVLA again showed strong leakage concentrated in the additive FFT step. A larger MLP was trained on 34,880 profiling traces and evaluated on 313,920 attack traces from 90 held-out keys, again achieving a 100% empirical recovery rate per trace.

4.3 A New Key Recovery Side-Channel Attack on HQC with Chosen Ciphertext

4.3.1 The RMRS Instantiation of HQC

This section review the work of Goy, Loiseau and Gaborit [17].
Recall from Section 2.3 that HQC is defined over the polynomial ring $\mathscr{R} = \mathbb{F}_q[x]/(x^n - 1)$, with $q = 2$ for the parameter sets standardized by NIST, and that a ciphertext has the form $c = (u(x), v(x))$ with

$$u(x) = r_1(x) + h(x)r_2(x), \qquad v(x) = m'(x) + s(x)r_2(x) + e(x),$$

where $m'(x) \in \mathscr{R}$ is the polynomial associated with the codeword $\mathbf{m}G \in \mathbb{F}_q^n$ of the public $[n,k]$-linear code $\mathscr{C}$, and where $s(x) = y(x) + h(x)z(x)$, with $y(x), z(x) \in \mathscr{R}$ the two components of the secret key, each of Hamming weight w. Decryption is performed by computing

$$c'(x) \;=\; v(x) - u(x)z(x) \;=\; m'(x) + y(x)r_2(x) - r_1(x)z(x) + e(x)$$

and by applying the decoding algorithm $\mathsf{Decode}_{\mathscr{C}}(\cdot)$ of $\mathscr{C}$ to $c'(x)$.

Unlike McEliece, where security relies on hiding the algebraic structure of a Goppa code, the code $\mathscr{C}$ used by HQC can be public, since security is reduced to the hardness of the quasi-cyclic syndrome decoding problem for the auxiliary parity-check matrix $H = (I_n \mid \mathsf{rot}(\mathbf{h}))$, and not to the secrecy of $\mathscr{C}$. This freedom allows the designers of HQC to choose $\mathscr{C}$ purely for decoding efficiency. The reference implementation instantiates $\mathscr{C}$ as the concatenation of:

- An external Reed–Solomon code, which is a $[n_1, k_1, d_1]$-linear code over $\mathbb{F}_{q_2}$ with $q_2 = 2^{k_2}$;
- An internal duplicated Reed–Muller code, which is a $[n_2, k_2, d_2]$-linear code over $\mathbb{F}_2$, applied independently to each of the n_1 symbols of $\mathbb{F}_{q_2}$ produced by the external encoder.

This construction, referred to as HQC-RMRS, encodes a message of length k_1 into a codeword of length $n_1 n_2$: the external RS code maps it to n_1 symbols of $\mathbb{F}_{q_2}$, and each symbol is independently re-encoded with the internal RM code into a block of length n_2. Decoding proceeds in the opposite order: the received word is first split into n_1 independent blocks of length n_2, each decoded with the internal RM decoder, and the n_1 resulting symbols of $\mathbb{F}_{q_2}$ are then fed to the external RS decoder.

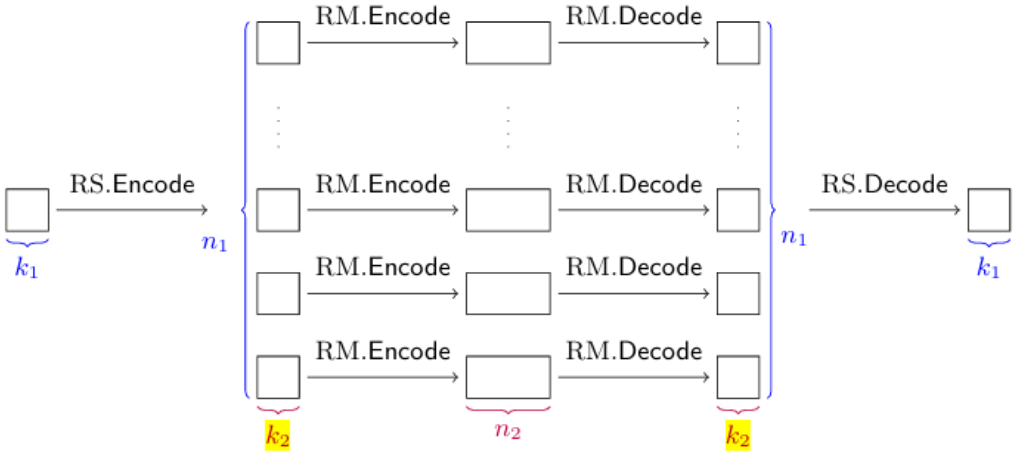


Fig. 4.1 Simplified HQC Concatenated RMRS Codes Framework [17]

Reed–Muller Codes and the Hadamard Transform

All three parameter sets of HQC use the same internal code, $\mathsf{RM}(1,7)$, a $[128,8,64]$-linear code over $\mathbb{F}_2$, whose codewords are repeated $\mathsf{mul} \in \{3,5\}$ times to obtain the effective block length $n_2 \in \{384,640\}$.

A first-order Reed–Muller code $RM(1,m)$ is combinatorially equivalent to a Hadamard code, and can be decoded using the Fast Hadamard Transform (FHT). The Hadamard matrix $W_{2^m} \in \{-1,1\}^{2^m \times 2^m}$ is described recursively by

$$W_{2^m} = \begin{pmatrix} W_{2^{m-1}} & W_{2^{m-1}} \\ W_{2^{m-1}} & -W_{2^{m-1}} \end{pmatrix}, \qquad W_1 = 1.$$

The FHT function correspond to multiplying the expand codeword with an Hadamard matrix. This allows finding of the distribution of the cosets, and consequently computing the distance between a received message and every codeword, resulting in a maximum lieklihood strategy for decoding.

A direct matrix–vector multiplication involving W_{2^m} would require $2^m \times 2^m$ additions and subtractions, resulting in a considerable computational cost. However, the Hadamard matrix can be decomposed into a product of m sparse $2^m \times 2^m$ matrices,

each containing only two non-zero elements per column:

$$W_{2^m} = M_{2^m}^{(1)} M_{2^m}^{(2)} \cdots M_{2^m}^{(m)}, \qquad M_{2^m}^{(i)} = I_{2^{m-i}} \otimes W_2 \otimes I_{2^{i-1}}, \quad 1 \leq i \leq m,$$

where $\otimes$ is the Kronecker product. Therefore, the computation can be performed iteratively through m stages rather than by directly multiplying by the complete Hadamard matrix.

The concrete decoding procedure applied to a received block $\mathbf{c} \in \mathbb{N}^{n_2}$ (with $n_2 = 128 \cdot \mathsf{mul}$) consists of three steps:

- **Expand and sum.** The mul repetitions of the $[128, 8, 64]$ codeword are collapsed into a single vector $\mathbf{c}' \in \mathbb{N}^{128}$ by

$$c'_j = \sum_{i=0}^{\mathsf{mul}-1} c_{128i+j}, \qquad j = 0, \ldots, 127;$$

- **Fast Hadamard Transform.** $\mathbf{c}'$ is transformed into $\widehat{\mathbf{c}} = \mathbf{c}' W_{128}$ by applying the FHT. This transformation returns a vector of length 128;

- **Find peaks.** The decoded message is recovered with the find peak function. Starting from the return vector of the FHT, the argument of the absolute maximum ($i^* = \arg\max_i |\widehat{c}_i|$) gives the 7 least significant bits of the decoded message. Finally, the sign of $\widehat{c}_{i^*}$ fixes the most significant bit of the decoded symbol.

The chapter's attack exploits the electromagnetic leakage of the FHT step applied during the decapsulation of an adversarially chosen ciphertext.

4.3.2 The Threat Model

The adversary $\mathscr{A}$ is assumed to have physical access to a device that repeatedly executes the HQC decapsulation algorithm Decaps with a fixed, static secret key $\mathsf{sk} = (y(x), z(x))$. $\mathscr{A}$ can submit arbitrary ciphertexts $c = (u(x), v(x))$ of its choice and, for each submitted ciphertext, records a trace T of the electromagnetic emanation of the device during the execution of the internal FHT routine. $\mathscr{A}$ does not need to observe the value returned by Decaps, which by construction of the underlying

Fujisaki–Okamoto-type transform reveals nothing about sk whether decapsulation succeeds or fails. Instead, the attack uses the leakage produced before the re-encryption check, i.e. during the computation of $c'(x) = v(x) - u(x)z(x)$ and its decoding, and is therefore unaffected by the implicit-rejection mechanism. The leakage trace T is then used to create an Oracle which is able to differentiate between the decoding patterns.

The goal of $\mathscr{A}$ is to recover the secret component $z(x)$. Once $z(x)$ is known, the companion component $y(x)$ follows immediately from the public relation $y(x) = s(x) - h(x)z(x)$, so recovering $z(x)$ is equivalent to a full break of the static key.

Support Distribution of the Secret Key

The attack chooses the special ciphertext $u(x) = 1$, $v(x) = 0$. Since subtraction and addition coincide over $\mathbb{F}_2$, the corresponding decoding input is

$$c'(x) \;=\; v(x) - u(x)z(x) \;=\; z(x).$$

The vulnerable decoder is thus made to decode the secret polynomial $z(x)$ itself. Because $z(x)$ has Hamming weight $w = \mathsf{wt}_H(z(x))$ much smaller than n (Table 2.7 lists $w \in \{66, 100, 131\}$ for the three security levels), each of the n_1 blocks fed to the internal RM decoder consists overwhelmingly of zero symbols, with only a handful of ones scattered according to the support of $z(x)$. This sparsity is precisely what the side-channel Oracle exploits: it allows the decoding behaviour of a block to be classified according to the number of ones it contains.

Since n is chosen as the smallest primitive prime larger than $n_1 n_2$, the last $\ell = n - n_1 n_2$ coordinates of $z(x)$ play no role in decoding and can be treated separately. Write

$$z(x) = \big(z'(x), z''(x)\big), \qquad z'(x) \in \mathbb{F}_2^{n_1 n_2}, \quad z''(x) \in \mathbb{F}_2^{\ell},$$

The decoding process only requires $z'(x)$, and the extra bits $z''(x)$ can be safely set to zero. In practice, the weight of $z''(x)$ is almost always zero, and even when it is not, it remains very small. If $w = \mathsf{wt}(z(x))$, the probability that $\mathsf{wt}(z''(x)) = k$ is approximately

$$\Pr[\mathsf{wt}(z''(x)) = k] \approx \binom{w}{k} p^k (1-p)^{w-k},$$

where $p = \frac{|z'(x)|}{|z(x)|}$ is the proportion of bits belonging to the useful part.

Since $\mathsf{wt}(z''(x))$ is typically zero, we have $\mathsf{wt}(z'(x)) \approx w$ in most cases. This is the worst-case scenario for the analysis, as it maximizes the weights of the blocks of $z'(x)$.
The vector $z'(x)$ can be furthered split into the n_1 blocks handled independently by the RM decoder,

$$z'(x) = \big(z'_0(x), \ldots, z'_{n_1-1}(x)\big), \qquad z'_i(x) \in \mathbb{F}_2^{n_2}.$$

The Reed-Muller decoder processes each block independently. For a randomly chosen block $z'(x)_i$, the probability that it has Hamming weight k is denoted by Pr_k.

$$\mathsf{Pr}_k \;:=\; \Pr\left[\mathsf{wt}_H(z'_i(x)) = k \;\middle|\; z \xleftarrow{\$} R_w,\; i \xleftarrow{\$} \{0, \ldots, n_1 - 1\}\right].$$

The probability is concentrated on small values of k: for all three HQC parameter sets, more than 99% of the probability mass lies on blocks of weight $k \leq 4$, and blocks of weight $k \geq 5$ occur only rarely. This observation motivates grouping all blocks of weight 5 or more into a single class, so that a block can be classified into one of $\tau + 1 = 6$ classes $\{0, 1, 2, 3, 4, \geq 5\}$.
This distribution is key for the distinguisher: it can only reliably identify blocks with weight up to a certain threshold τ. Since the majority of blocks fall into this range, the attack remains effective under the typical (and worst-case) assumption that $\mathsf{wt}(z''(x)) = 0$.

A secondary effect must also be accounted for: the expand-and-sum step adds the mul repeated copies of a block over $\mathbb{N}$, so if two of the w nonzero positions of $z(x)$ happen to coincide modulo 128 within the same block, the corresponding entry of $\mathbf{c}'$ takes the value 2 instead of 1, which is a higher-magnitude error (HME) that slightly alters the FHT leakage compared to a block of the same Hamming weight without collisions. Denoting by $\mathsf{Pr}_{\mathsf{HME}}$ the probability that a randomly sampled block of weight k contains such a collision, this probability turns out to be below 1% for all HQC parameter sets, thus not disrupting significantly the Oracle.

4.3.3 The Attack Algorithm

The Decoding Oracle

Definition 46 (RM Decoding Oracle). Let $i \in \{0,\dots,n_1-1\}$ and let $\tau \geq 0$ be a fixed threshold. The Oracle $\mathscr{O}^{RM}_{i,b}$ is a probabilistic algorithm that takes as input an HQC ciphertext $c = (u(x), v(x))$, together with side-channel leakage recorded during the decapsulation of c, and outputs a value $b \in \{0,1,\dots,\tau\}$, equal, with high probability, to the number of errors corrected by the internal RM decoder in the i-th block of $c'(x) = v(x) - u(x)z(x)$, provided that this number does not exceed τ.

For the special ciphertext of the previous section, $c'(x) = z(x)$ and the decoder corrects (almost always) exactly $\mathsf{wt}_H(z'_i(x))$ errors in block i, so $\mathscr{O}^{RM}_{i,b}$ applied to $(1,0)$ returns a reference value equal to the weight of the i-th block of the secret key.

Recovering the Support of a Block

Fix a block index j and let $\mathbf{v}_j \in \mathbb{F}_2^{n_2}$ denote a candidate perturbation of Hamming weight 1 applied to the j-th block of $v(x)$ (all other blocks of $u(x), v(x)$ being left at the reference choice $u(x) = 1, v(x) = 0$). The corresponding decoding input for block j becomes $\mathbf{v}_j - z'_j(x)$, and two cases arise, distinguishable by querying $\mathscr{O}^{RM}_{j,b}$:

- If the single nonzero position of $\mathbf{v}_j$ lies in $\mathsf{Supp}(z'_j(x))$, the two errors cancel and
$$\mathscr{O}^{RM}_{j,b}(\mathbf{v}_j - z'_j(x)) \;=\; \mathscr{O}^{RM}_{j,b}(z'_j(x)) - 1;$$
- Otherwise, the perturbation adds a new error and
$$\mathscr{O}^{RM}_{j,b}(\mathbf{v}_j - z'_j(x)) \;=\; \mathscr{O}^{RM}_{j,b}(z'_j(x)) + 1.$$

Testing all n_2 possible weight-one vectors $\mathbf{v}_j$ therefore reveals the entire support $\mathsf{Supp}(z'_j(x))$: a decrease of one unit relative to the reference value identifies a support position, an increase rules it out.

Divide-and-Conquer Strategy

Since the n_1 blocks of the RM decoder are processed independently, the n_1 oracles $\mathcal{O}^{RM}_{0,b}, \ldots, \mathcal{O}^{RM}_{n_1-1,b}$ can be queried simultaneously by perturbing one position in every block at once and recording a single trace, rather than perturbing one block at a time. This reduces the number of distinct queries required to recover the full support of $z'(x)$ from $n_1 n_2$ down to n_2 -multiplied by the number s of repeated traces needed for the side-channel Oracle to reach a reliable decision on a single query.
Once $\mathsf{Supp}(z'(x))$ has been recovered, the residual ℓ-bit tail $z''(x)$ can be recovered either by exhaustive search, since its expected weight is close to zero, or, more generally, in polynomial time using the support-completion technique of Schamberger et al [18].

4.3.4 Experimental Results

Experimental Setup

The Oracle $\mathcal{O}^{RM}_{i,b}$ is trained on electromagnetic traces acquired from an ARM Cortex-M4 microcontroller (clock frequency 168 MHz) running the reference implementation of the FHT routine. Traces are acquired with an EM near field microprobe and a 750 MHz sampling rate oscilloscope, with an externally supplied clock used to remove jitter across acquisitions and a GPIO trigger placed immediately before and after the FHT call, so that each trace corresponds exactly to the execution of the FHT code on a single block.

A training set is built by sampling, for each of the six classes $k \in \{0, 1, 2, 3, 4, \geq 5\}$ introduced above, a population of blocks of the corresponding Hamming weight (with the proportion of higher-magnitude-error blocks matched to the theoretical value of P_{HME}), and by recording 10 000 traces per class.

Leakage Assessment

The separability of the six classes is first quantified with a Welch t-test. A sample point of the trace is considered leaking if $|t|$ exceeds the conventional threshold of 4.5. Applying this test to every pair among the six classes shows a strong distinguishability concentrated in two regions of interest (RoI) of the trace. Restricting the classifier to

these two regions both improves accuracy and drastically reduces the dimensionality, and hence the computational cost, of the subsequent classification step.

Classification and Results

The Oracle $\mathcal{O}_{i,b}^{RM}$ is realized as a Linear Discriminant Analysis (LDA) classifier trained on the selected regions of interest. Since a single trace does not, in general, allow the classifier to reach a reliable decision, s independent traces of the same query are combined by summing the class-probability vectors returned by the classifier for each trace and combining the traces with a soft-max technique. With a training set of 40 000 traces per class, $s = 50$ attack traces per query are sufficient to reach essentially perfect single-bit recovery.

Combined with the divide-and-conquer strategy, this yields a total attack cost of $s \times n_2$ electromagnetic traces to recover the full support of $z'(x)$. For the HQC-128 parameter set ($n_2 = 384$), this amounts to $50 \times 384 = 19\,200$ traces, i.e. fewer than 20 000 measurements to fully recover the static secret key used for decapsulation.

Countermeasure

The distinguisher exploited above relies on the FHT being computed directly on the (mostly sparse) block $\mathbf{c}'$ derived from the secret key. A natural defence against the attack is to apply a mask. The goal is to hide sensitive intermediate values by splitting them into multiple random shares. Since the Fast Hadamard Transform (FHT) is a linear operation, if the input $\mathbf{c}$ can be written as

$$\mathbf{c} = \sum_{i=0}^{n} \mathbf{c}_i,$$

then, by linearity, the transform satisfies

$$\mathrm{FHT}(\mathbf{c}) = \sum_{i=0}^{n} \mathrm{FHT}(\mathbf{c}_i).$$

This property allows the computation of the transform on each share separately and then combine the results. In order to build a secure masking scheme, the n shares

$\mathbf{c}_0, \ldots, \mathbf{c}_{n-1}$ are randomly sampled, and the last share is defined as

$$\mathbf{c}_n = \mathbf{c} - \sum_{i=0}^{n-1} \mathbf{c}_i,$$

so that the sum of all shares equals the original input $\mathbf{c}$.

An attacker attempting to recover information from the masked computation would need to target all shares simultaneously, since each share is uniformly random and independent. The individual shares do not satisfy the low Hamming weight property that the distinguisher relies upon. The randomization destroys the correlation between the share's weight and the secret class, making the statistical test ineffective.

To assess the strength of this countermeasure, experiments were conducted on a large number of masked expanded codewords. The Welch t-test was used to detect differences between classes. The results showed a significant reduction in detectable statistical leakage compared to the unmasked case. This indicates that the mask provides a strong barrier against the proposed attack.

Chapter 5

Side Chanel Analysis of McEliece and HQC Key Encapsulation Mechanisms

5.1 Motivations and Objectives of the Analysis

Chapter 4 reviewed two published key-recovery side-channel attacks, one against Classic McEliece and one against the RMRS instantiation of HQC, both of which exploit an hardware leakage: a specific internal computation is profiled with power or electromagnetic traces and a trained classifier, then the resulting distinguisher is queried tens of thousands of times to recover the static secret key. Both attacks require an attacker with physical or near-physical access to the target device, dedicated acquisition hardware, and a profiling phase carried out on a clone device under the attacker's control.

Those attacks needed a profiled power or EM setup to see their leakage at all. This chapter asks a simpler question first: is any leakage still visible from the outside, in the plain CPU-cycle count of a whole Decaps call, with no power probe and no trained classifier? The experiment implemented uses the TVLA methodology and Welch t-test of Section 4.1.1 not to confirm a known leak, but as a cheap first check that any implementer of these PQClean builds could run before trusting them.

Concretely, the objective of this chapter is to evaluate, on the same reference implementations and the same experimental platform used for the performance benchmark of Chapter 3, whether the decapsulation routine of Classic McEliece

(across its five "f" parameter sets, from mceliece348864f through mceliece8192128f) and of HQC (across all three NIST security levels, hqc-128, hqc-192 and hqc-256) exhibits a CPU-cycle-count dependence on two input-related conditions:

1. **Fixed vs. random ciphertext.** The canonical TVLA configuration, in which one class of measurements repeatedly decapsulates the same ciphertext while the other decapsulates a freshly generated one on every trial;
2. **Valid vs. corrupted ciphertext.** A second experiment in which every ciphertext is freshly generated and valid, but a subset of the measurements is first tampered with by flipping a controlled number of bits before decapsulation. In particular, two corruption intensities are taken into consideration: a single flipped bit and a heavy corruption flipping half of the ciphertext's bits, so as to probe both the boundary of the decoder's error-correction capability and a regime in which decoding is expected to fail outright.

The choice of testing only the "f" variants of McEliece is choerent with the fact that the experiments take into account for measurements only the decapsulation process, where there is no difference between the two algorithm variants.

5.2 Analysis Environment and Experimental Methodology

The analysis presented in this chapter is conducted on the same PQClean reference implementations, x86/x86-64 target, and low-level environmental controls introduced for the performance benchmark of Section 3.2.

5.2.1 Reuse of the Environmental Controls

Because the leakage under test is a difference of at most a few percent in mean cycle count between two classes of measurements, the analysis is even more sensitive to uncontrolled sources of timing noise. For this reason, every environmental safeguard introduced during the benchmark experiment is retained without modification: the measurement process is pinned to a single logical core to avoid cross-core migration

and cache-state churn, its scheduling priority is raised on a best-effort basis to reduce preemption by other processes, and a CPU warm-up routine is executed before any timed measurement to rule out frequency-scaling artefacts.
In place of the per-algorithm key-generation warm-up used for the benchmark, here a short sequence of decapsulation calls on the fixed ciphertext is performed immediately before each test begins.

5.2.2 Choice of Timing Instrumentation

A single metric is collected for every timed decapsulation: the raw CPU cycle count obtained via the `rdtsc` instruction, using the same primitive and the same justification (hardware-independence, immunity to NTP-induced slewing, sub-microsecond resolution) discussed in Section 3.2.3. Unlike the benchmark, wall-clock nanosecond timestamps are not additionally collected here: since the goal of this chapter is exclusively statistical leakage detection rather than a cross-algorithm performance comparison, a single, stable channel is preferable.

5.2.3 Experiment 1: Fixed vs. Random Ciphertext

The first experiment reproduces the canonical TVLA configuration: a key pair and a single "fixed" ciphertext are generated once at the start of the campaign, and every subsequent trial is randomly assigned to one of two classes, decapsulating either that same fixed ciphertext or a freshly generated one. The class of each trial is drawn independently and uniformly at random rather than measured in two contiguous blocks of "fixed" and "random" trials: interleaving the two classes is a standard TVLA precaution to prevent the class from being biased by slow changes in the measurement environment, such as residual frequency scaling or gradual thermal drift during a long test.

5.2.4 Experiment 2: Valid vs. Corrupted Ciphertext

The second experiment is designed to isolate a different, and arguably more relevant, source of potential leakage: not whether a ciphertext is reused across trials, but whether it lies within the code's declared error-correction radius. To this end, a new,

valid ciphertext is generated independently for every trial of both classes, removing the fixed-vs-fresh asymmetry of Experiment 1 entirely; the two classes differ only in whether that ciphertext is decapsulated unmodified (class "valid"), or is first corrupted by flipping a controlled number of pseudorandomly chosen, distinct bit positions before decapsulation is timed (class "corrupted"). This directly probes whether the decoder's execution profile differs once its input departs from the code's correction radius.
Two corruption intensities are used, chosen to cover the two qualitative regimes of interest: a single flipped bit, which probes the decoder at the boundary of its error-correction capability, where a small, localized difference in the decoding work performed is most plausible; and a heavy corruption flipping half of the ciphertext's bits, which pushes the decoder well outside its correction radius, into a regime where decoding is expected to fail unambiguously and where any leakage tied to detecting or handling failure would be expected to be most pronounced. The latter is reported as the "half corrupted bit" experiment in the results of Section 5.4.

5.2.5 Statistical Treatment of the Measurements

Every trial of both experiments produces a labelled CPU-cycle sample, collected into per-class arrays of 10000 measurements. As a first, in-process sanity check, a Welch t-statistic is computed on the raw samples at the end of each test, against the same $|t| > 4.5$ TVLA threshold, and reported alongside the mean cycle count of each class. The definitive statistical analysis is however performed offline, on the raw per-trial samples that each campaign serialises to a CSV file, by a pararell analysis routine that recomputes the Welch t-test twice: once on the raw samples, and once on samples independently trimmed, per class, at the 1^{st} and 99^{th} percentile. This trimming step is motivated by the same concern already raised in previously about scheduler- and kernel-induced noise: sporadic long-latency interruptions (context switches, page faults, background OS activity) generate long right tails in the cycle-count distribution that are unrelated to the cryptographic computation under test and can, if left untreated, either mask a genuine difference between classes or manufacture a spurious one.

5.3 Description of the Test Suite and of the Code Structure

The test suite for this chapter is organized into two parts: a C measurement harness, which tests the PQClean implementations under the two experimental designs explained above and collects raw CPU-cycle samples, and a Python post-processing script, which turns those raw samples into the statistical data and figures presented in Section 5.4.

5.3.1 Reused Helper and Infrastructure Functions

The CPU-affinity and process-priority setters, set_cpu_affinity and set_high_priority, and the CPU warm-up routine warmup_cpu, are carried over unchanged from the benchmark harness of Chapter 3 (Listings 3.1–3.2 and the surrounding discussion in Section 3.3.1) and are not reproduced here. Furthermore, the cycle-counting primitive get_cpu_cycles is the same `rdtsc`-based function shown in Listing 3.5.

5.3.2 Ciphertext Corruption

The only new piece of low-level infrastructure introduced for this chapter is corrupt _ciphertext (Listing 5.10), which implements the bit-flipping step required by Experiment 2. Given a ciphertext buffer, its length in bytes, and a target number of bits to flip, the function allocates a Boolean "used" bitmap covering every bit position of the buffer, and repeatedly draws a pseudorandom bit index by combining two calls to rand(), rejecting any index already marked as used, until exactly the requested number of distinct bit positions have been toggled via XOR.
Working with distinct positions, rather than simply drawing n_bits independent indices with possible repeats, guarantees that the requested corruption intensity, one bit or half of the ciphertext, is met exactly rather than only in expectation.

```
86  void corrupt_ciphertext(unsigned char *buf, size_t len, int n_bits) {
87      size_t total_bits = len * 8;
88      if ((size_t)n_bits > total_bits)
89          n_bits = (int)total_bits;
90
91      unsigned char *used = calloc(total_bits, 1);
92      int flipped = 0;
93      while (flipped < n_bits) {
94          size_t bit_idx = ((size_t)rand() * (size_t)rand()) %
            ↪  total_bits;
95          if (used[bit_idx]) continue;
96          used[bit_idx] = 1;
97          buf[bit_idx / 8] ^= (1 << (bit_idx % 8));
98          flipped++;
99      }
100     free(used);
101 }
```

Listing 5.10 Function used to corrupt a ciphertext by flipping a controlled number of distinct random bits

5.3.3 Measurement Routines

The core measurement logic is implemented by two functions, run_timing_test and run_corrupted_test, each corresponding to one of the two experiments motivated above.

run_timing_test (Listing 5.11) implements Experiment 1. After generating a single key pair and a single fixed ciphertext, it allocates two arrays of n_measurements doubles to hold the per-class cycle samples, then enters the main loop: on each iteration a class is drawn uniformly at random, and either the fixed ciphertext or a freshly encapsulated one is decapsulated with its execution bracketed by two calls to get_cpu_cycles. Once the loop completes, the function computes the mean cycle count of each class and an in-process Welch t-statistic for an immediate console verdict, and finally serializes every raw labelled sample to a two-column CSV file (class, cycles) for the offline analysis described below.

```
139 for (int i = 0; i < n_measurements; i++) {
140     int cls = rand() % 2;
141
142     if (cls == 0) {
143         // Class 0: fixed ciphertext
144         uint64_t start = get_cpu_cycles();
145         algo->decaps(ss_tmp, ct_fixed, sk);
146         uint64_t end = get_cpu_cycles();
147         cycles_fixed[n_fixed++] = (double)(end - start);
148     } else {
149         // Class 1: fresh ciphertext
150         algo->encaps(ct_random, ss_tmp, pk);
151         uint64_t start = get_cpu_cycles();
152         algo->decaps(ss_tmp, ct_random, sk);
153         uint64_t end = get_cpu_cycles();
154         cycles_random[n_random++] = (double)(end - start);
155     }
156
157     if (i % 500 == 0) { printf("."); fflush(stdout); }
158 }
```

Listing 5.11 Core measurement loop of function run_timing_test (fixed-vs-random ciphertext)

run_corrupted_test (Listing 5.12) implements Experiment 2, and takes two additional parameters compared to run_timing_test: n_bits_flip, the number of bits to corrupt, and a string label used to tag the output CSV file. Unlike Experiment 1, a fresh, valid ciphertext is encapsulated on every iteration, for both classes; the randomly drawn class then determines only whether that ciphertext is decapsulated unmodified, or is first passed, on a copy, to corrupt_ciphertext (Listing 5.10) before its decapsulation is timed. As in run_timing_test, the function concludes by reporting the mean cycle count of each class, an in-process Welch t-statistic, and by writing every raw labelled sample to a CSV file named after the algorithm and the corruption label (”1bit” or ”heavy”).

```
for (int i = 0; i < n_measurements; i++) {
    // Fresh valid ciphertext for each measurement
    algo->encaps(ct_valid, ss_tmp, pk);

    int cls = rand() % 2;
    // Class 0: valid ciphertext
    if (cls == 0) {
        uint64_t start = get_cpu_cycles();
        algo->decaps(ss_tmp, ct_valid, sk);
        uint64_t end = get_cpu_cycles();
        cycles_valid[n_valid++] = (double)(end - start);
    } else {
        // Class 1: corrupted ciphertext
        memcpy(ct_corrupt, ct_valid, algo->ct_size);
        corrupt_ciphertext(ct_corrupt, algo->ct_size, n_bits_flip);

        uint64_t start = get_cpu_cycles();
        algo->decaps(ss_tmp, ct_corrupt, sk);
        uint64_t end = get_cpu_cycles();
        cycles_corrupt[n_corrupt++] = (double)(end - start);
    }

    if (i % 500 == 0) { printf("."); fflush(stdout); }
}
```

Listing 5.12 Core measurement loop of function run_corrupted_test (valid-vs-corrupted ciphertext)

5.3.4 Main Function

The main (Listing 5.13) applies the environmental control, seeds the pseudorandom number generator from the wall clock, declares the KemAlgo descriptor for the parameter set under test, and then invokes run_timing_test once, followed by run_corrupted_test twice, with n_bits_flip set respectively to 1 and to ct_size $\times$ 4, i.e. half of the ciphertext's total bits. Every parameter set reported in Section 5.4 is obtained by recompiling and rerunning this same source with the corresponding header include and KemAlgo initializer swapped in for that scheme.

```
KemAlgo mceliece = {
    .name = "mceliece8192128f",
    .keypair =
    ↪ (keypair_fn)PQCLEAN_MCELIECE8192128F_CLEAN_crypto_kem_keypair,
    .encaps = (enc_fn)PQCLEAN_MCELIECE8192128F_CLEAN_crypto_kem_enc,
    .decaps = (dec_fn)PQCLEAN_MCELIECE8192128F_CLEAN_crypto_kem_dec,
    .pk_size = PQCLEAN_MCELIECE8192128F_CLEAN_CRYPTO_PUBLICKEYBYTES,
    .sk_size = PQCLEAN_MCELIECE8192128F_CLEAN_CRYPTO_SECRETKEYBYTES,
    .ct_size = PQCLEAN_MCELIECE8192128F_CLEAN_CRYPTO_CIPHERTEXTBYTES,
    .ss_size = PQCLEAN_MCELIECE8192128F_CLEAN_CRYPTO_BYTES,
};
```

Listing 5.13 KemAlgo descriptor for a parameter set, here Classic McEliece mceliece8192128f

```
// Test 1: fixed vs random ciphertexts
run_timing_test(&mceliece, NUM_MEASUREMENTS);

// Test 2: valid vs corrupted ciphertexts with 1 bit flipped
run_corrupted_test(&mceliece, NUM_MEASUREMENTS, 1, "1bit");

// Test 3: valid vs corrupted ciphertexts with heavy corruption (4
↪ bits per byte)
run_corrupted_test(&mceliece, NUM_MEASUREMENTS, (int)(mceliece.ct_size
↪ * 4), "heavy");
```

Listing 5.14 Test-campaign orchestration in main for a single parameter set

5.3.5 Offline Analysis Script

The CSV files written by run_timing_test and run_corrupted_test are consumed by the parallel script analyze_timing_csv.py, whose statistical role was already motivated in Section 5.2.5; only its code organization is summarised here, without reproducing it. The script is structured around two functions: analyze_file loads one CSV, validates that it contains exactly the two expected classes, produces the percentile-trimmed samples, and computes and prints both the raw and the trimmed

statistics together with the leak/no-leak verdict; and plot_result renders, for each analyzed file, the overlaid density histograms and box plot presented alongside the corresponding table in Section 5.4. A main function iterates the first two of these over every CSV file passed on the command line and prints a final tabular summary across all analyzed files. Every table and figure presented in Section 5.4 is produced directly by this script, run once per CSV file generated by the corresponding experiment and parameter set.

5.4 Results

This section reports, for every parameter set of Classic McEliece (only the "f" variant) and HQC introduced in Table 3.1, the outcome of the two experiments described previously: the fixed-vs-random ciphertext test, and the valid-vs-corrupted ciphertext test at both the single-bit and the half-of-the-ciphertext corruption intensity. For each algorithm and each experiment, the average and standard deviation of the CPU-cycle measurements are reported for both classes, together with the Welch t-statistic and p-value computed on the raw data and on the 1^{st}–99^{th} percentile trimmed data (as produced by analyze_timing_csv.py), alongside the corresponding timing-distribution and box-plot figures. A $|t|$-value exceeding the 4.5 threshold is taken, as in Chapter 4, to indicate a statistically detectable leak; all other cases are reported as showing no leak detectable at this statistical power.

5.4.1 Classic McEliece

mceliece348864f

Mode	**Mean (cycles)**	**Std (cycles)**
Fixed	52228695.6	1241089.6
Random	52206606.4	1145137.6

Data	**t-Welch**	**p-value**
Raw data	0.925	3.55×10^{-1}
Trimmed data	0.596	5.51×10^{-1}

Table 5.1 Statistics for the fixed vs random ciphertext experiment for McEliece348864f

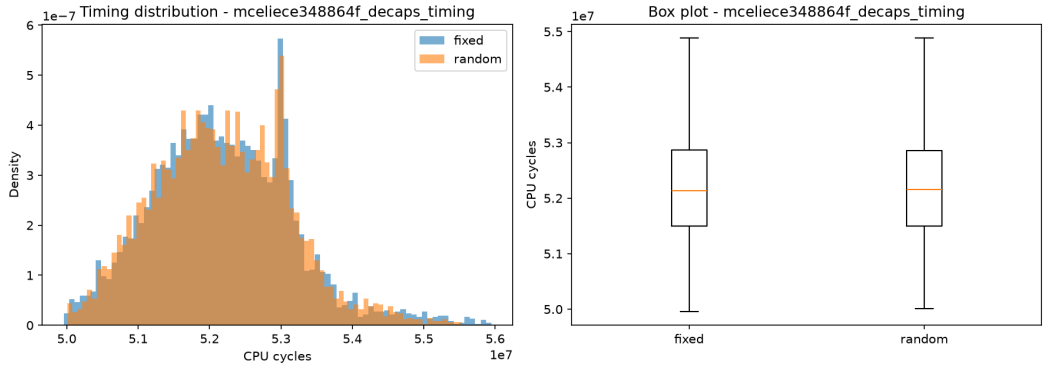


Fig. 5.1 Graphic results for the fixed vs random ciphertext experiment for McEliece348864f

Mode	**Mean (cycles)**	**Std (cycles)**
Valid	52271931.6	1065467.7
Corrupted	52274979.5	1075993.2

Data	***t*-Welch**	***p*-value**
Raw data	-0.142	8.87×10^{-1}
Trimmed data	-0.115	9.09×10^{-1}

Table 5.2 Statistics for the valid vs corrupted ciphertext with 1 corrupted bit experiment for McEliece48864f

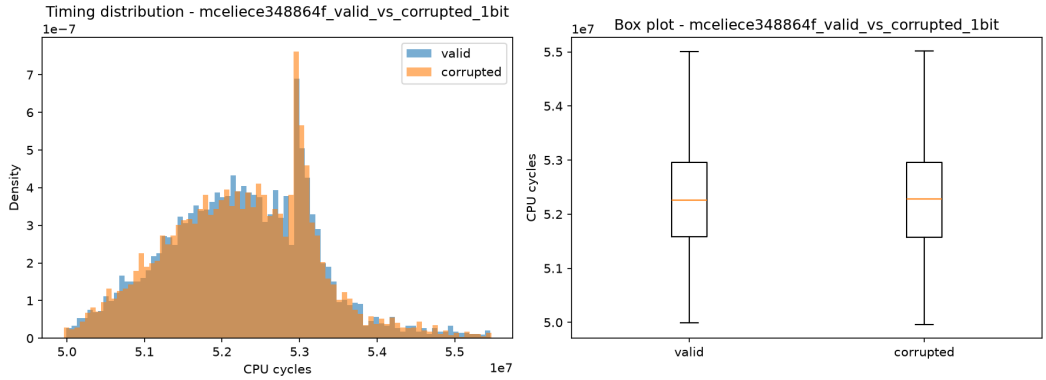


Fig. 5.2 Graphic results for the valid vs corrupted ciphertext with 1 corrupted bit experiment for McEliece348864f

Mode	Mean (cycles)	Std (cycles)
Valid	52291474.7	1136894.8
Corrupted	52292529.9	1109003.2

Data	t-Welch	p-value
Raw data	-0.047	9.63×10^{-1}
Trimmed data	-0.485	6.28×10^{-1}

Table 5.3 Statistics for the valid vs corrupted ciphertext with half corrupted bit experiment for McEliece348864f

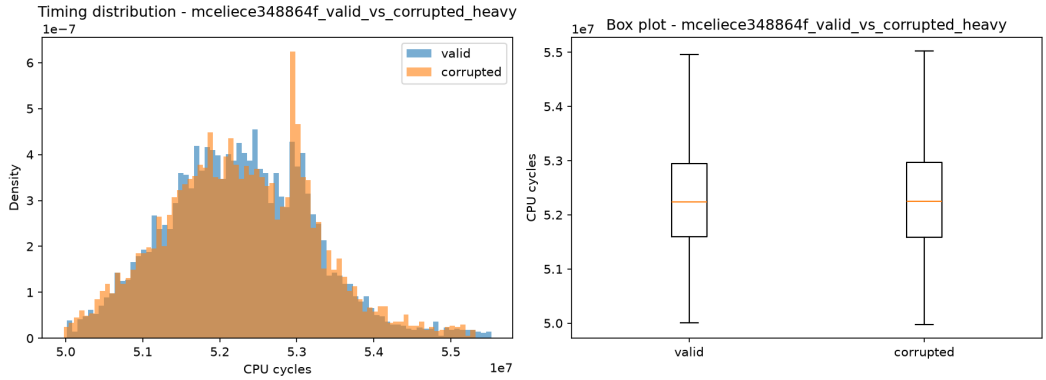


Fig. 5.3 Graphic results for the valid vs corrupted ciphertext with half corrupted bit experiment for McEliece348864f

mceliece460896f

Mode	Mean (cycles)	Std (cycles)
Fixed	105196984.7	2375897.6
Random	105143068.2	2280637.4

Data	t-Welch	p-value
Raw data	1.158	2.47×10^{-1}
Trimmed data	1.238	2.16×10^{-1}

Table 5.4 Statistics for the fixed vs random ciphertext experiment for McEliece460896f

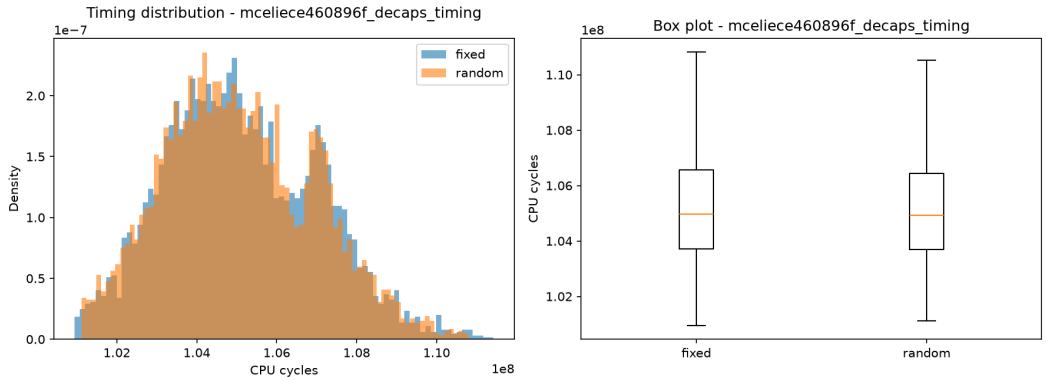


Fig. 5.4 Graphic results for the fixed vs random ciphertext experiment for McEliece460896f

Mode	**Mean (cycles)**	**Std (cycles)**
Valid	105044054.2	2043735.4
Corrupted	105106020.3	2130621.4

Data	***t*-Welch**	***p*-value**
Raw data	-1.484	1.38×10^{-1}
Trimmed data	-1.513	1.30×10^{-1}

Table 5.5 Statistics for the valid vs corrupted ciphertext with 1 corrupted bit experiment for McEliece460896f

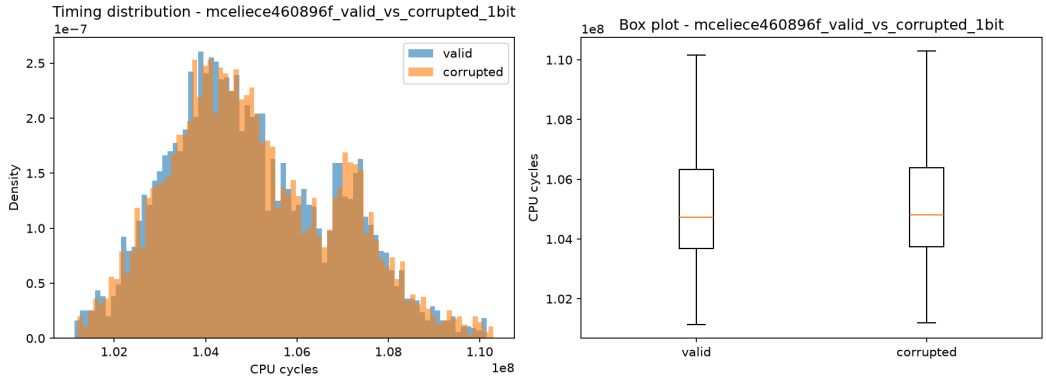


Fig. 5.5 Graphic results for the valid vs corrupted ciphertext with 1 corrupted bit experiment for McEliece460896f

Mode	Mean (cycles)	Std (cycles)
Valid	105742021.1	2441623.1
Corrupted	105722738.8	2404860.8

Data	t-Welch	p-value
Raw data	0.398	6.91×10^{-1}
Trimmed data	0.705	4.81×10^{-1}

Table 5.6 Statistics for the valid vs corrupted ciphertext with half corrupted bit experiment for McEliece460896f

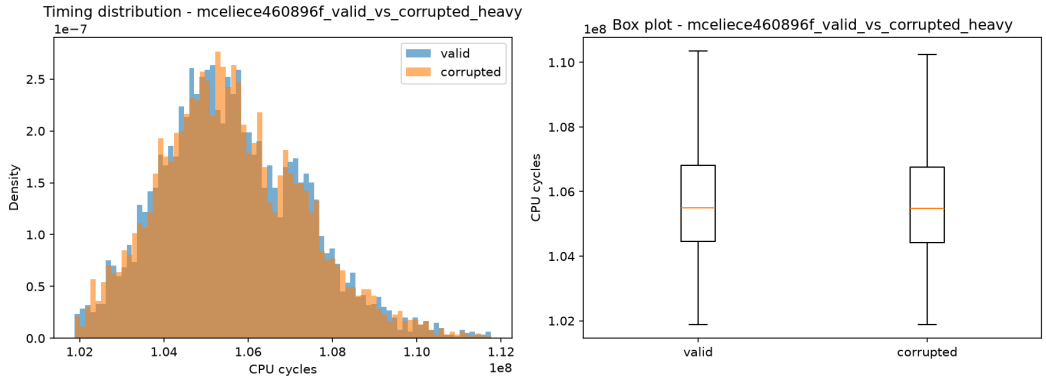


Fig. 5.6 Graphic results for the valid vs corrupted ciphertext with half corrupted bit experiment for McEliece460896f

mceliece6688128f

Mode	Mean (cycles)	Std (cycles)
Fixed	203465764.1	3630851.0
Random	203404192.4	3588903.0

Data	t-Welch	p-value
Raw data	0.853	3.94×10^{-1}
Trimmed data	1.154	2.49×10^{-1}

Table 5.7 Statistics for the fixed vs random ciphertext experiment for McEliece6688128f

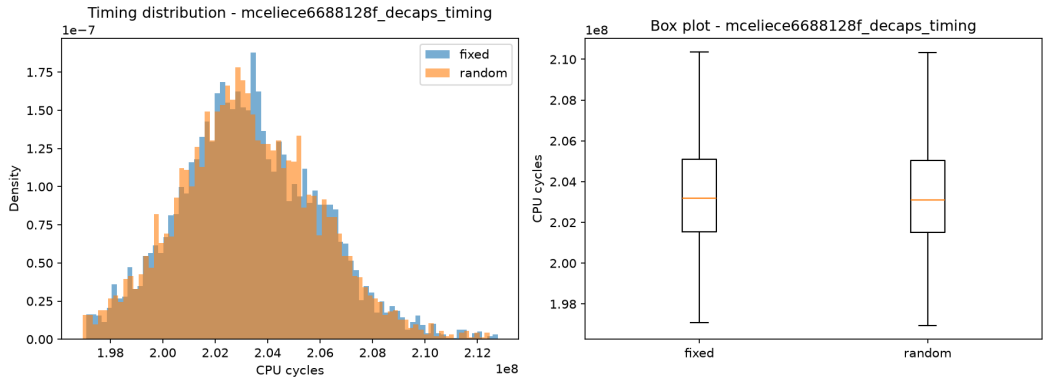


Fig. 5.7 Graphic results for the fixed vs random ciphertext experiment for McEliece6688128f

Mode	**Mean (cycles)**	**Std (cycles)**
Valid	202652870.4	3434121.2
Corrupted	202569983.6	3485101.4

Data	***t*-Welch**	***p*-value**
Raw data	1.198	2.31×10^{-1}
Trimmed data	1.603	1.09×10^{-1}

Table 5.8 Statistics for the valid vs corrupted ciphertext with 1 corrupted bit experiment for McEliece6688128f

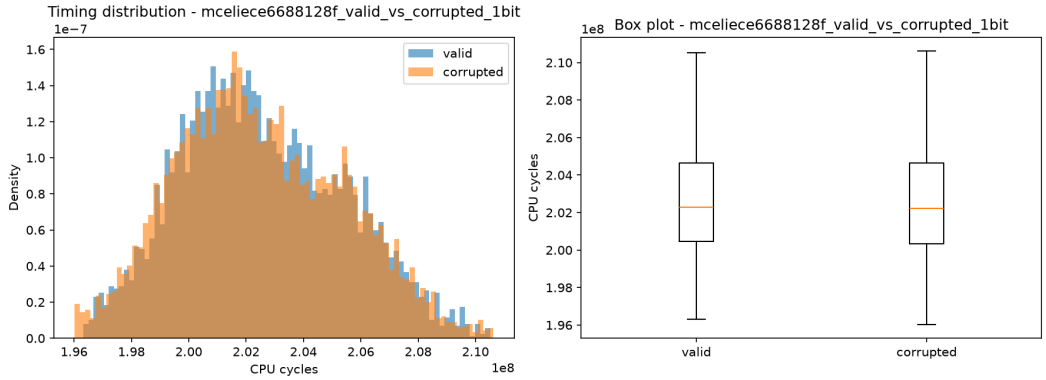


Fig. 5.8 Graphic results for the valid vs corrupted ciphertext with 1 corrupted bit experiment for McEliece6688128f

Mode	Mean (cycles)	Std (cycles)
Valid	202689619.8	3595768.0
Corrupted	202603452.7	3292386.8

Data	t-Welch	p-value
Raw data	1.251	2.11×10^{-1}
Trimmed data	0.806	4.20×10^{-1}

Table 5.9 Statistics for the valid vs corrupted ciphertext with half corrupted bit experiment for McEliece6688128f

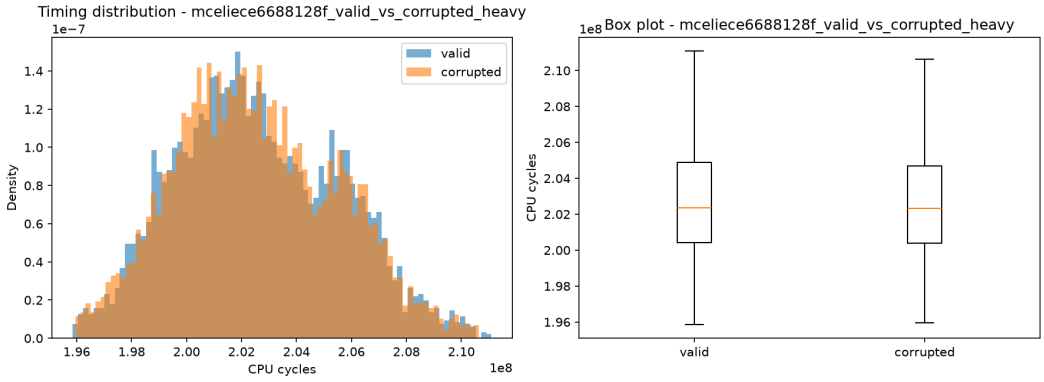


Fig. 5.9 Graphic results for the valid vs corrupted ciphertext with half corrupted bit experiment for McEliece6688128f

mceliece6960119f

Mode	Mean (cycles)	Std (cycles)
Fixed	195503551.0	3649290.0
Random	195593583.6	3514996.7

Data	t-Welch	p-value
Raw data	-1.257	2.09×10^{-1}
Trimmed data	-1.851	6.43×10^{-2}

Table 5.10 Statistics for the fixed vs random ciphertext experiment for McEliece6960119f

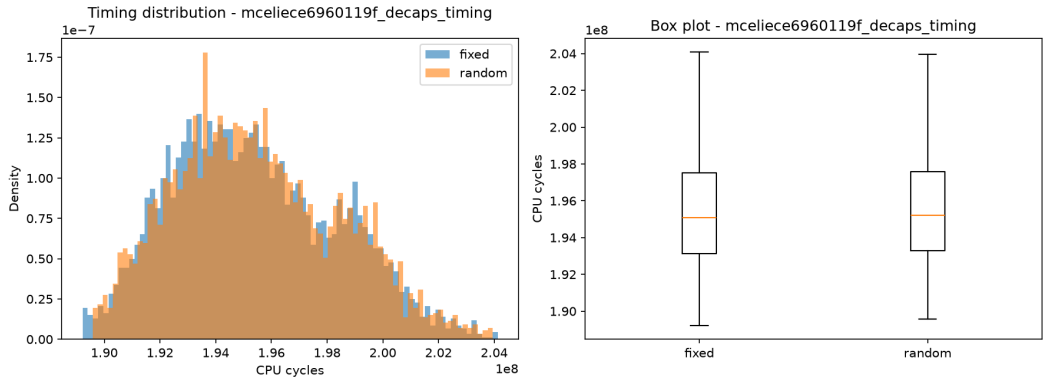


Fig. 5.10 Graphic results for the fixed vs random ciphertext experiment for McEliece6960119f

Mode	**Mean (cycles)**	**Std (cycles)**
Valid	196077053.9	3098331.6
Corrupted	196132233.7	3531553.9

Data	***t*-Welch**	***p*-value**
Raw data	-0.829	4.07×10^{-1}
Trimmed data	0.250	8.03×10^{-1}

Table 5.11 Statistics for the valid vs corrupted ciphertext with 1 corrupted bit experiment for McEliece6960119f

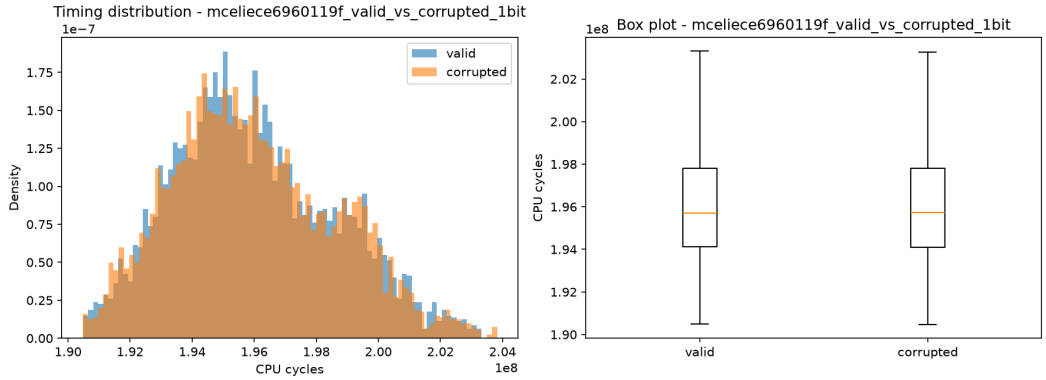


Fig. 5.11 Graphic results for the valid vs corrupted ciphertext with 1 corrupted bit experiment for McEliece6960119f

Mode	Mean (cycles)	Std (cycles)
Valid	196357108.0	3113249.1
Corrupted	196452349.0	3212113.7

Data	t-Welch	p-value
Raw data	-1.506	1.32×10^{-1}
Trimmed data	-1.485	1.38×10^{-1}

Table 5.12 Statistics for the valid vs corrupted ciphertext with half corrupted bit experiment for McEliece6960119f

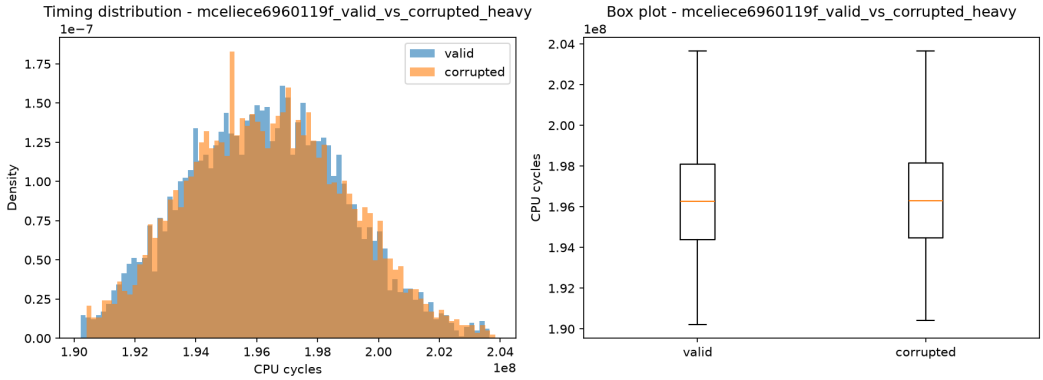


Fig. 5.12 Graphic results for the valid vs corrupted ciphertext with half corrupted bit experiment for McEliece6960119f

mceliece8192128f

Mode	Mean (cycles)	Std (cycles)
Fixed	248918774.0	4073313.5
Random	248870015.0	3911250.9

Data	t-Welch	p-value
Raw data	0.610	5.42×10^{-1}
Trimmed data	0.517	6.05×10^{-1}

Table 5.13 Statistics for the fixed vs random ciphertext experiment for McEliece8192128f

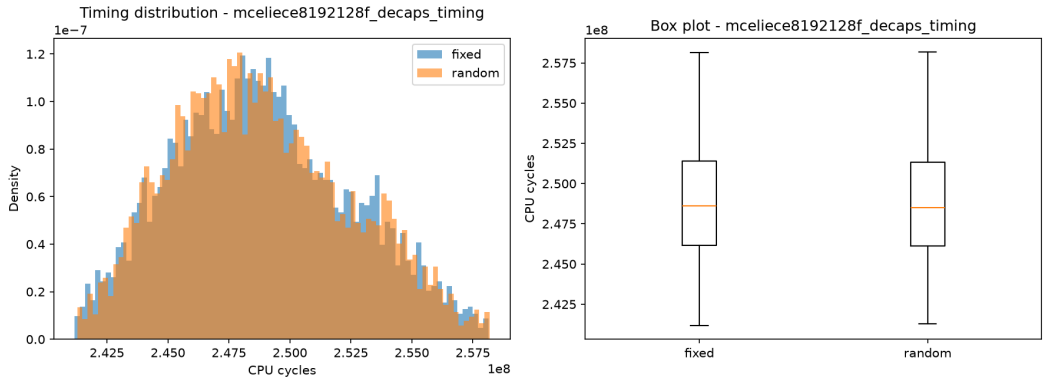


Fig. 5.13 Graphic results for the fixed vs random ciphertext experiment for McEliece8192128f

Mode	**Mean (cycles)**	**Std (cycles)**
Valid	248355499.6	4236256.9
Corrupted	248441700.3	4548110.5

Data	***t*-Welch**	***p*-value**
Raw data	-0.981	3.27×10^{-1}
Trimmed data	-0.668	5.04×10^{-1}

Table 5.14 Statistics for the valid vs corrupted ciphertext with 1 corrupted bit experiment for McEliece8192128f

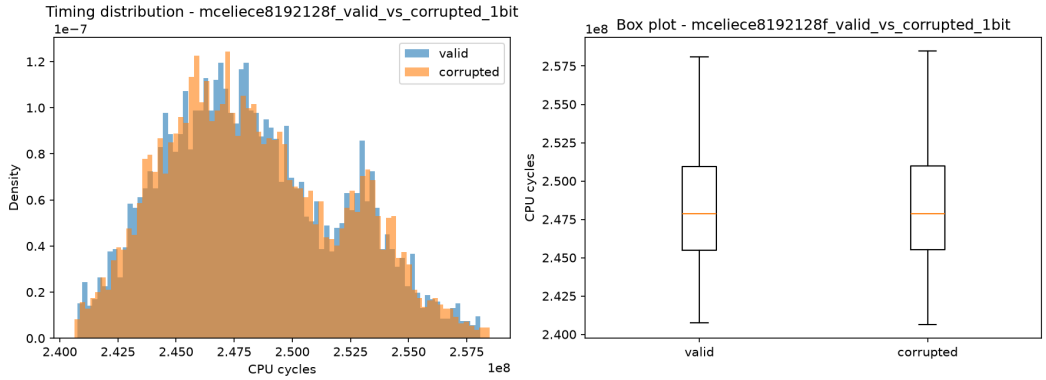


Fig. 5.14 Graphic results for the valid vs corrupted ciphertext with 1 corrupted bit experiment for McEliece8192128f

Mode	Mean (cycles)	Std (cycles)
Valid	249771516.9	3698638.5
Corrupted	249706618.0	3943980.1

Data	t-Welch	p-value
Raw data	0.849	3.96×10^{-1}
Trimmed data	1.384	1.66×10^{-1}

Table 5.15 Statistics for the valid vs corrupted ciphertext with half corrupted bit experiment for McEliece8192128f

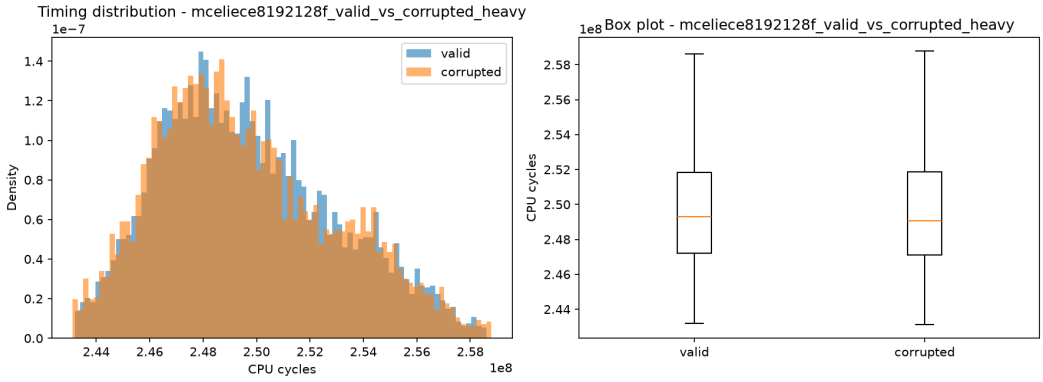


Fig. 5.15 Graphic results for the valid vs corrupted ciphertext with half corrupted bit experiment for McEliece8192128f

5.4.2 HQC

hqc- – 128

Mode	Mean (cycles)	Std (cycles)
Fixed	7687799.3	329782.1
Random	7692170.8	314270.7

Data	t-Welch	p-value
Raw data	-0.679	4.97×10^{-1}
Trimmed data	-1.283	2.00×10^{-1}

Table 5.16 Statistics for the fixed vs random ciphertext experiment for HQC-128

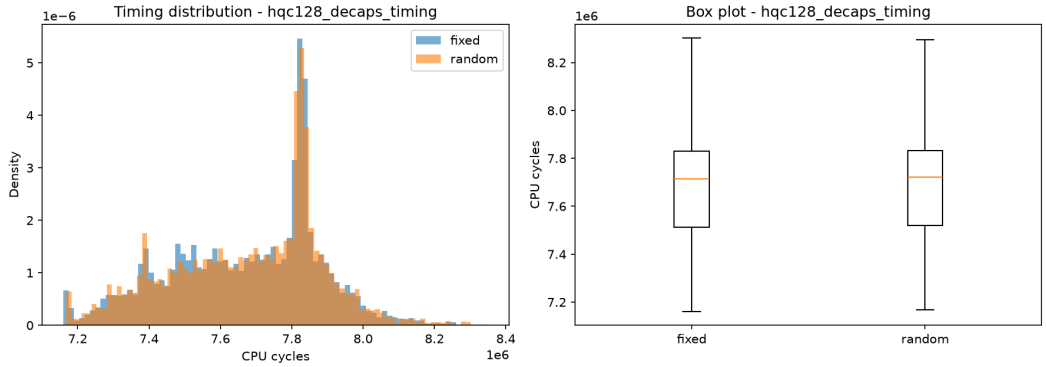


Fig. 5.16 Graphic results for the fixed vs random ciphertext experiment for HQC-128

Mode	**Mean (cycles)**	**Std (cycles)**
Valid	7778355.4	296249.8
Corrupted	7772365.8	278886.0

Data	***t*-Welch**	***p*-value**
Raw data	1.041	2.98×10^{-1}
Trimmed data	1.123	2.61×10^{-1}

Table 5.17 Statistics for the valid vs corrupted ciphertext with 1 corrupted bit experiment for HQC-128

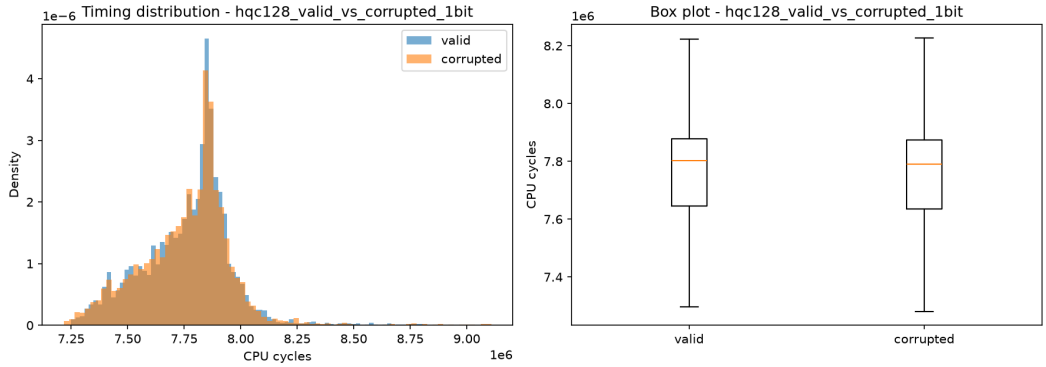


Fig. 5.17 Graphic results for the valid vs corrupted ciphertext with 1 corrupted bit experiment for HQC-128

Mode	Mean (cycles)	Std (cycles)
Valid	7793769.8	275174.1
Corrupted	7841176.2	286506.9

Data	t-Welch	p-value
Raw data	-8.439	3.65×10^{-17}
Trimmed data	-10.502	1.15×10^{-25}

Table 5.18 Statistics for the valid vs corrupted ciphertext with half corrupted bit experiment for HQC-128

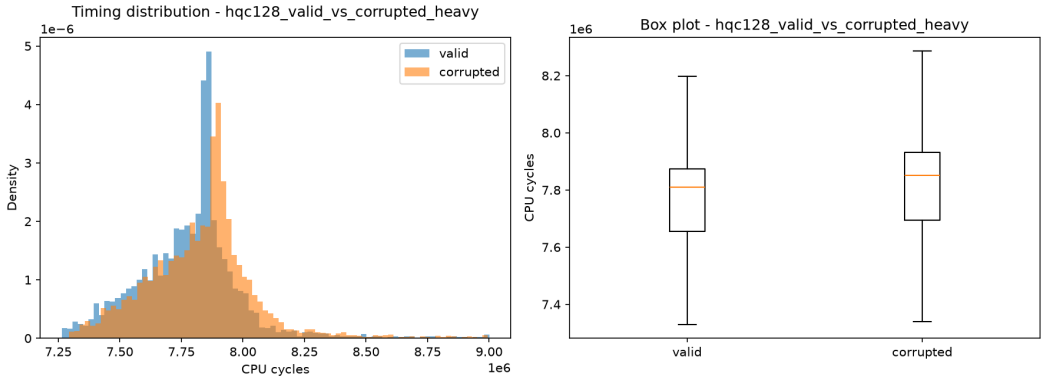


Fig. 5.18 Graphic results for the valid vs corrupted ciphertext with half corrupted bit experiment for HQC-128

hqc-192

Mode	Mean (cycles)	Std (cycles)
Fixed	23280897.6	586346.0
Random	23282037.9	593472.6

Data	t-Welch	p-value
Raw data	-0.097	9.23×10^{-1}
Trimmed data	-0.276	7.83×10^{-1}

Table 5.19 Statistics for the fixed vs random ciphertext experiment for HQC-192

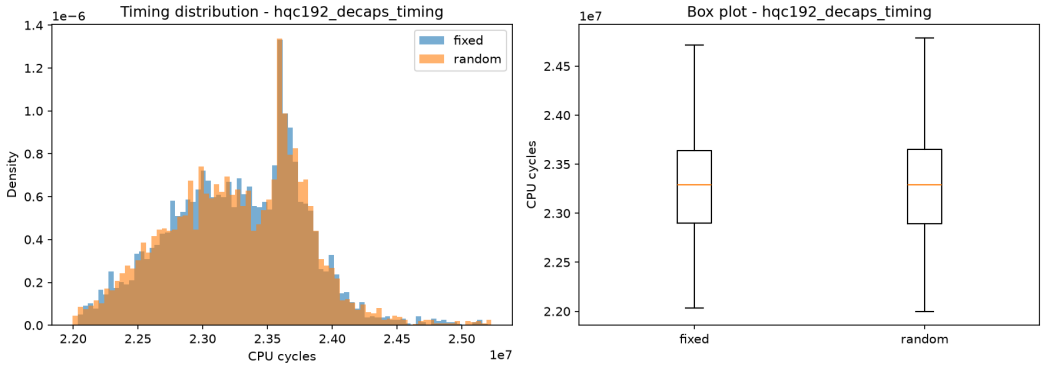


Fig. 5.19 Graphic results for the fixed vs random ciphertext experiment for HQC-192

Mode	**Mean (cycles)**	**Std (cycles)**
Valid	23360840.0	816691.9
Corrupted	23366215.2	708782.1

Data	***t*-Welch**	***p*-value**
Raw data	-0.352	7.25×10^{-1}
Trimmed data	-0.927	3.54×10^{-1}

Table 5.20 Statistics for the valid vs corrupted ciphertext with 1 corrupted bit experiment for HQC-192

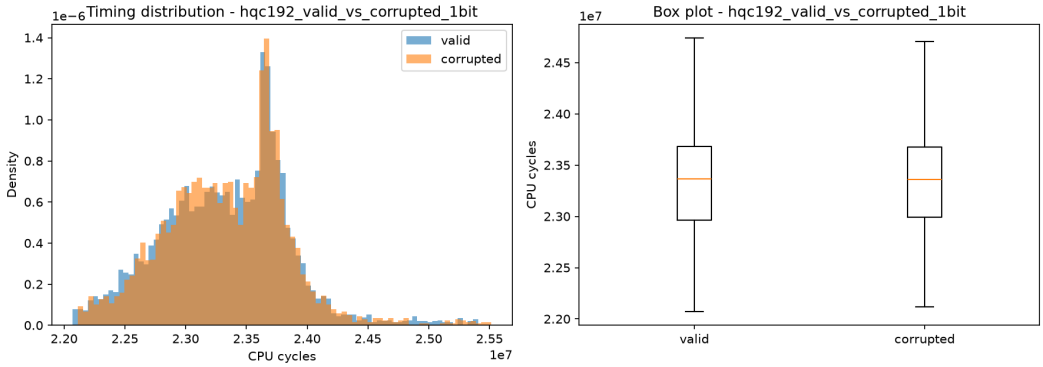


Fig. 5.20 Graphic results for the valid vs corrupted ciphertext with 1 corrupted bit experiment for HQC-192

Mode	Mean (cycles)	Std (cycles)
Valid	23539363.1	760237.3
Corrupted	23587431.0	609285.2

Data	t-Welch	p-value
Raw data	-3.487	4.90×10^{-4}
Trimmed data	-6.628	3.57×10^{-11}

Table 5.21 Statistics for the valid vs corrupted ciphertext with half corrupted bit experiment for HQC-192

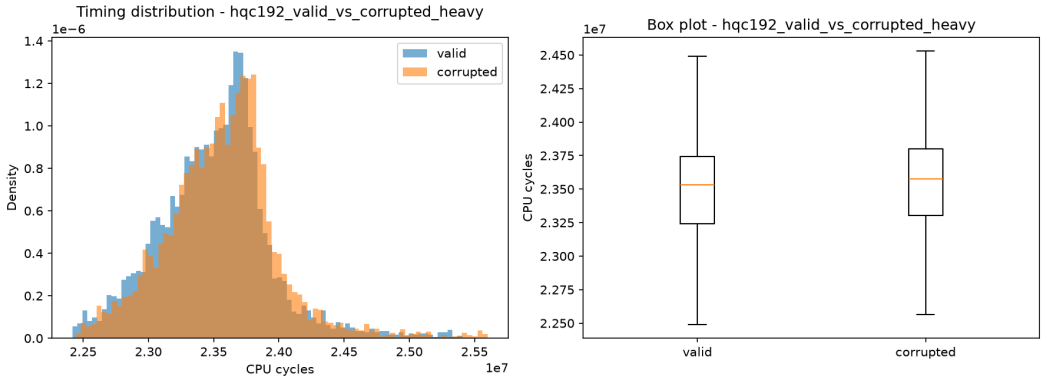


Fig. 5.21 Graphic results for the valid vs corrupted ciphertext with half corrupted bit experiment for HQC-192

hqc-256

Mode	Mean (cycles)	Std (cycles)
Fixed	44283107.6	1409909.5
Random	44287257.0	1437061.5

Data	t-Welch	p-value
Raw data	-0.146	8.84×10^{-1}
Trimmed data	-0.455	6.49×10^{-1}

Table 5.22 Statistics for the fixed vs random ciphertext experiment for HQC-256

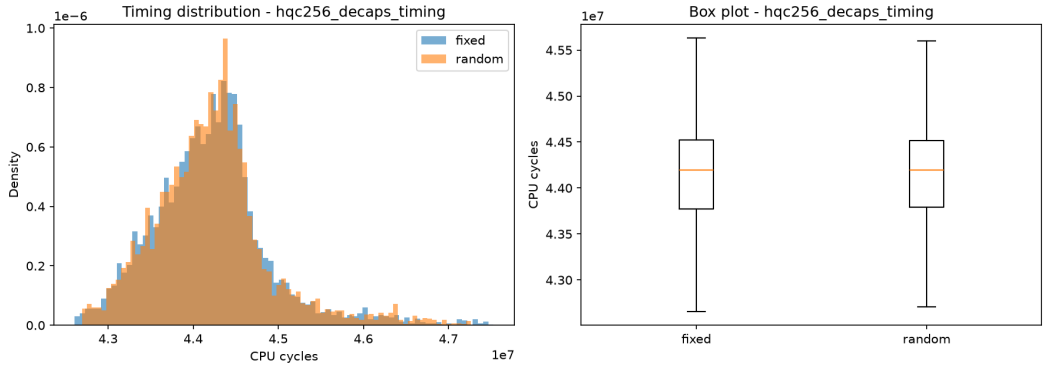


Fig. 5.22 Graphic results for the fixed vs random ciphertext experiment for HQC-256

Mode	**Mean (cycles)**	**Std (cycles)**
Valid	44123428.2	1003074.8
Corrupted	44135525.5	1167789.7

Data	***t*-Welch**	***p*-value**
Raw data	-0.556	5.79×10^{-1}
Trimmed data	-0.132	8.95×10^{-1}

Table 5.23 Statistics for the valid vs corrupted ciphertext with 1 corrupted bit experiment for HQC-256

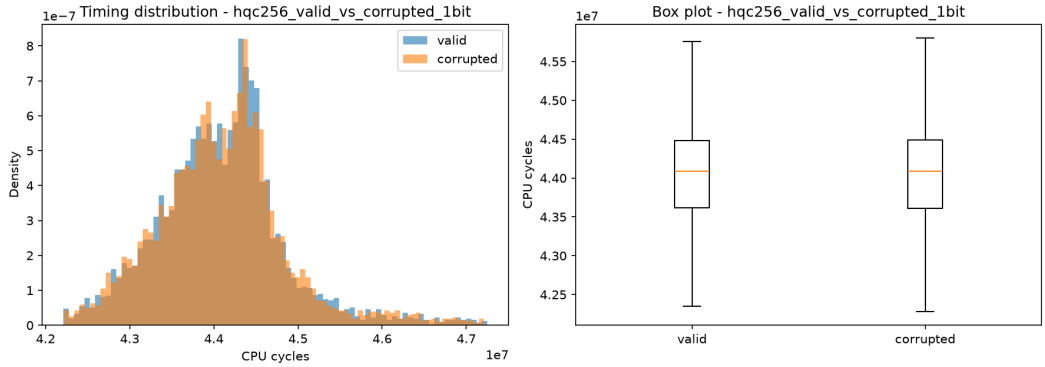


Fig. 5.23 Graphic results for the valid vs corrupted ciphertext with 1 corrupted bit experiment for HQC-256

Mode	Mean (cycles)	Std (cycles)
Valid	43837820.6	1098752.9
Corrupted	43873601.6	1151856.3

Data	t-Welch	p-value
Raw data	-1.590	1.12×10^{-1}
Trimmed data	-1.915	5.55×10^{-2}

Table 5.24 Statistics for the valid vs corrupted ciphertext with half corrupted bit experiment for HQC-256

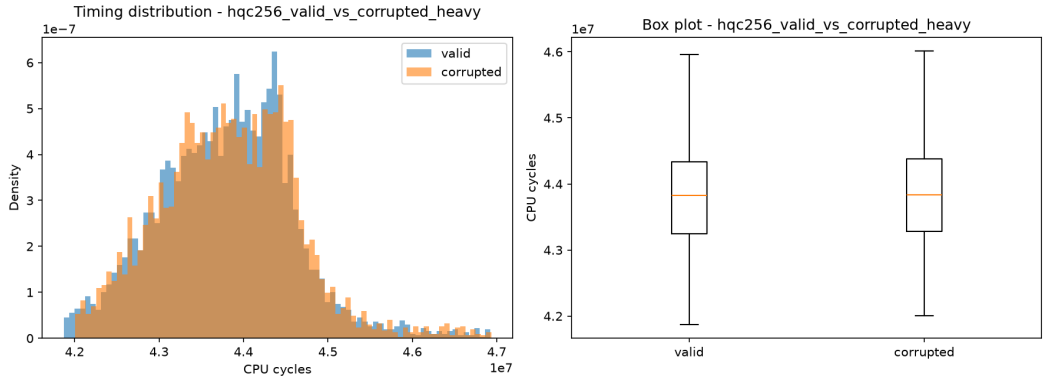


Fig. 5.24 Graphic results for the valid vs corrupted ciphertext with half corrupted bit experiment for HQC-256

5.5 Conclusions

5.5.1 Classic McEliece

Across all five "f" parameter sets, and all three experiments, the Classic McEliece results are uniform: not a single trimmed $|t|$-value comes close to the 4.5 threshold. The largest magnitude observed is $|t| = 1.851$, for mceliece6960119f in the fixed-vs-random experiment (Table 5.10), and every other configuration sits below $|t| \approx 1.6$. This holds equally for the fixed-vs-random test, for a single corrupted bit, and for half of the ciphertext corrupted, with no trend that grows with security level. In other words, the cycle count of Decaps gives no indication, at this sample size, that Classic McEliece behaves any differently when handed a fixed ciphertext, a corrupted single bit, or a ciphertext with half its bits flipped.

This absence of any detectable signal is consistent with what a direct inspection of the PQClean reference source shows. The decoding pipeline exercised by Decaps, namely decrypt.c, bm.c (Berlekamp–Massey) and root.c (the additive-FFT polynomial evaluation), is written without a single data-dependent branch: every loop runs for a fixed number of iterations fixed by the compile-time constants SYS_T and SYS_N, regardless of the ciphertext or of how many errors it actually encodes, and the final decoding-success check is computed with bitwise arithmetic (check $-=$ 1; check $>>=$ 15;) rather than with an if statement. With the control flow itself fixed by construction, and with a very small computational footprint per call, ciphertexts of 96–208 bytes, any residual microarchitectural effect of the kind discussed below for HQC would have to be extremely small in absolute terms, and the results here indicate that, if present at all, it stays below the noise floor achievable with $n = 10\,000$ samples on this platform.

5.5.2 HQC

The picture for HQC is different, but only partially. The fixed-vs-random experiment and the single-bit corruption experiment behave exactly like McEliece: every $|t|$-value stays below 1.3 across all three security levels (Tables 5.16–5.17, 5.19–5.20, 5.22–5.23), giving no indication of leakage. The half-corrupted-bit experiment, however, tells a different story at the lower security levels. For hqc-128, the trimmed Welch statistic reaches $t = -10.502$ ($p = 1.15 \times 10^{-25}$, Table 5.18), and the raw-data statistic already exceeds the threshold on its own ($t = -8.439$). For hqc-192, the trimmed statistic also crosses the threshold, $t = -6.628$ ($p = 3.57 \times 10^{-11}$, Table 5.21), although here the raw-data statistic alone ($t = -3.487$) would not have been enough, underlining the value of the outlier-trimming step. For hqc-256, finally, the same experiment falls just short of the threshold ($t = -1.915$, Table 5.24), so no leak is flagged at the highest security level.

Before interpreting this pattern, it is worth stressing what a direct inspection of the hqc-256 reference source rules out. Just like Classic McEliece, every routine on the decapsulation path is written without data-dependent branches. reed_solomon.c computes the error-locator polynomial in compute_elp with a loop that always runs $2 \cdot$ PARAM_DELTA times, updating its internal state through bitmasks rather than through conditionals; reed_muller.c locates the decoded symbol in find_peaks with the same masked-selection idiom instead of an if (x > max) comparison, and is

explicitly documented in-source as a "constant time implementation"; and the outer key-encapsulation wrapper in kem.c performs the Fujisaki–Okamoto re-encryption check and the resulting choice between the real and the implicit-rejection shared secret with a masked selection, $\mathsf{mc[i]} = (\mathsf{m[i]}\ \&\ \mathsf{result}) \wedge (\mathsf{sigma[i]}\ \&\ {\sim}\mathsf{result})$, again with no branch on secret- or ciphertext-dependent data. With the control-flow path of the reference implementation identical regardless of how many errors the received word actually contains, the measured cycle difference cannot be explained by a classical "more iterations for more errors" leak of the kind such masking is designed to prevent; it must instead originate below the level the C source can control, most plausibly in microarchitectural effects, such as data-dependent instruction latency or dynamic frequency response to the different switching activity, that a branch-free implementation does not, and cannot, eliminate.

This is also consistent with which experiment shows the effect and which does not: a single corrupted bit perturbs at most one of the PARAM_N1 Reed–Muller blocks and leaves the reference implementation's uniform control flow, and its overall data profile, almost unchanged, so no signal is expected and none is observed ($|t| < 1.3$ at every level). Corrupting half of the ciphertext's bits, by contrast, changes the values processed by a large fraction of the decoding pipeline at once; the resulting difference in switching activity or memory traffic, small at the level of a single operation, only becomes large enough to be statistically visible once it is aggregated over that much larger amount of altered data, which is exactly the regime the heavy corruption experiment creates and the single-bit one does not.

The decline of $|t|$ from Level 1 to Level 5 does not, on its own, indicate that the underlying effect disappears. The absolute difference between the valid and the corrupted class means in the heavy corruption experiment is essentially flat across security levels, $\approx 47\,400$ cycles at hqc-128 (Table 5.18), $\approx 48\,100$ cycles at hqc-192 (Table 5.21), and $\approx 35\,800$ cycles at hqc-256 (Table 5.24), even though PARAM_N1, PARAM_N2 and the mean decapsulation cost itself grow by a factor of 3 to nearly $6\times$ over the same range (from under 8 million cycles at Level 1 to over 43 million cycles at Level 5). The standard deviation of the measurements, however, grows together with the mean decapsulation cost, from $\approx 280\,000$ cycles at Level 1 to ≈ 1.1 million cycles at Level 5, roughly a $4\times$ increase, because a longer-running call gives the scheduler, the cache hierarchy and any residual frequency transients more opportunities to introduce jitter, even under the environmental controls. Since the Welch t-statistic is a signal-to-noise ratio, a roughly constant numerator divided

by a roughly quadrupling denominator falls by close to the same factor, which is exactly the decline observed from $t \approx 10.5$ at Level 1 to $t \approx 6.6$ at Level 3 to $t \approx 1.9$ at Level 5. The sample size $n = 10\,000$, kept fixed across all three levels for comparability with the rest of this chapter, is therefore adequate to resolve the effect at the lower security levels but is no longer large enough to separate it from the growing measurement noise at hqc-256.

5.5.3 Conclusions

Taken together, the results of this chapter draw a clear distinction between the two schemes at the level of a software-based timing test. Classic McEliece shows no detectable dependence of its decapsulation cycle count on the ciphertext across any of the three experiments and any of the five parameter sets tested, consistent both with its branch-free reference implementation and with its comparatively small per-call computational footprint. HQC, in contrast, seems to exhibits a highly significant timing difference once a ciphertext is corrupted heavily enough to alter a large fraction of the data processed by its RMRS decoder, even though its reference implementation is, at the level of control flow, just as branch-free as McEliece's; the effect is strongest at the lowest security level and becomes statistically indistinguishable from measurement noise by the time the highest security level is reached, an outcome traced here to the measurement noise growing faster than the underlying signal as the parameter set grows, rather than to the signal itself vanishing. A single corrupted bit, at any security level, produces no detectable difference for either scheme.

Chapter 6

Conclusions and Future Work

This thesis analized the code-based public-key encryption and key encapsulation from three complementary angles: the theoretical foundations of Classic McEliece and HQC, an empirical benchmark of their reference implementations, and an analysis of their resistance to side-channel leakage. This chapter draws these threads together and discusses their implications.

Classic McEliece and HQC embody two different design philosophies. McEliece hides the structure of a binary Goppa code behind a scrambling transformation, so breaking it means solving a generic-looking instance of syndrome decoding, while HQC keeps its code public and instead hides the message under an error term too heavy for that code's own decoder.
This structural difference translates directly into opposite performance profiles. At every security level tested, HQC offers smaller keys, a smaller combined ciphertext-plus-key size, lower total time, less memory usage, and better measurement stability, while McEliece offers faster encapsulation and smaller ciphertexts. The reason is algorithmic: McEliece's encapsulation is a single cheap matrix-vector product, but its decapsulation requires a full syndrome-decoding routine, and its key generation involves Gaussian elimination on a matrix that grows with the code length n; HQC instead spreads a moderate polynomial-ring cost across both encapsulation and decapsulation. The gap in total time was also shown to widen in absolute terms as security increases, since McEliece's key-generation cost scales faster than HQC's.

Within the McEliece family, the results indicated that the error-correction weight t matters more for performance than the code length n alone, and that the "f" parameter

sets should be preferred over the standard ones in essentially all cases, since accepting a semi-systematic parity-check form removes the frequent key-generation restarts that make the standard variants both slower and less stable.

The two literature attacks reviewed in this thesis showed that both schemes' reference implementations can be broken by a physically profiled attacker: an additive-FFT attack against McEliece and a chosen-ciphertext attack exploiting Reed–Muller decoding against HQC both achieved near-complete key recovery using power or electromagnetic traces. Both, however, require a profiled measurement setup and physical or near-physical access to the target device.

The original software-level analysis of this thesis asked a more conservative question: is any leakage still visible using only the plain CPU-cycle count of a decapsulation call, with no power probe and no trained classifier? The answer differed between the two schemes. Classic McEliece showed no detectable dependence of its decapsulation time on the ciphertext under any tested condition, consistent with its fully branch-free decoding pipeline. HQC behaved the same way except under heavy ciphertext corruption, where a statistically significant timing difference emerged, strongest at the lowest security level and no longer distinguishable from noise at the highest. Since the HQC reference code is equally free of data-dependent branches, this residual signal cannot come from the control flow itself; it more plausibly reflects microarchitectural effects, such as data-dependent instruction latency, that only become visible once a large fraction of the ciphertext is altered. Notably, the decline in significance across security levels reflects growing measurement noise rather than a vanishing effect, since the absolute cycle-count difference between valid and corrupted ciphertexts stayed roughly constant across all three levels.

The benchmark and the side-channel analysis were both conducted on a single platform using unmodified PQClean reference implementations, and the sample size used, while adequate at lower security levels, was shown to be insufficient to resolve the HQC leakage at the highest one. Future work should repeat these experiments on different hardware and with larger sample sizes, investigate the precise microarchitectural origin of the HQC timing signal, and extend the comparison to optimized or constant-time-hardened implementations and to other code-based NIST candidates.

References

[1] Violetta Weger, Niklas Gassner, and Joachim Rosenthal. A survey on code-based cryptography. *arXiv preprint arXiv:2201.07119*, 2022.

[2] Emmanuel Abbe, Ori Sberlo, Amir Shpilka, Min Ye, et al. *Reed-muller codes*. Now Publishers, 2023.

[3] Elwyn Berlekamp, Robert McEliece, and Henk Van Tilborg. On the inherent intractability of certain coding problems (corresp.). *IEEE Transactions on Information theory*, 24(3):384–386, 1978.

[4] Nicolas Sendrier and Dimitrios E Simos. How easy is code equivalence over fq? In *International Workshop on Coding and Cryptography-WCC 2013*, 2013.

[5] Carlo Sanna. Cryptographic assumptions, public-key encryption part i and ii. Lecture slides, course of *Advanced Cryptography*, Politecnico di Torino, A.A. 2024/2025, 2025.

[6] Eiichiro Fujisaki and Tatsuaki Okamoto. Secure integration of asymmetric and symmetric encryption schemes. In *Annual international cryptology conference*, pages 537–554. Springer, 1999.

[7] Peter W Shor. Algorithms for quantum computation: discrete logarithms and factoring. In *Proceedings 35th annual symposium on foundations of computer science*, pages 124–134. Ieee, 1994.

[8] Ugochukwu Orji. An overview of quantum cryptography and shor's algorithm. *International Journal of Advanced Trends in Computer Science and Engineering*, 2020.

[9] Lily Chen, Lily Chen, Stephen Jordan, Yi-Kai Liu, Dustin Moody, Rene Peralta, Ray A Perlner, and Daniel Smith-Tone. *Report on post-quantum cryptography*, volume 12. US Department of Commerce, National Institute of Standards and Technology . . . , 2016.

[10] National Institute of Standards and Technology. Security (evaluation criteria), 2026.

[11] Robert J McEliece. A public-key cryptosystem based on algebraic. *Coding Thv*, 4244(1978):114–116, 1978.

[12] Tung Chou, Carlos Cid, S UiB, J Gilcher, T Lange, V Maram, R Misoczki, R Niederhagen, K Paterson, and E Persichetti. Classic mceliece: conservative code-based cryptography, 10 october 2020, 2020.

[13] Philippe Gaborit, Carlos Aguilar-Melchor, Nicolas Aragon, Slim Bettaieb, Loïc Bidoux, Olivier Blazy, Jean-Christophe Deneuville, Edoardo Persichetti, Gilles Zémor, Jurjen Bos, et al. Hamming quasi-cyclic (hqc). *NIST Post-Quantum Standardization, 4th Round*, 2025.

[14] Matthias J. Kannwischer, Peter Schwabe, Douglas Stebila, and Thom Wiggers. Improving software quality in cryptography standardization projects. In *IEEE European Symposium on Security and Privacy, EuroS&P 2022 - Workshops, Genoa, Italy, June 6-10, 2022*, pages 19–30, Los Alamitos, CA, USA, 2022. IEEE Computer Society.

[15] Paul C Kocher. Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems. In *Annual international cryptology conference*, pages 104–113. Springer, 1996.

[16] Qian Guo, Andreas Johansson, and Thomas Johansson. A key-recovery side-channel attack on classic mceliece implementations. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 800–827, 2022.

[17] Guillaume Goy, Antoine Loiseau, and Philippe Gaborit. A new key recovery side-channel attack on hqc with chosen ciphertext. In *International Conference on Post-Quantum Cryptography*, pages 353–371. Springer, 2022.

[18] Thomas Schamberger, Julian Renner, Georg Sigl, and Antonia Wachter-Zeh. A power side-channel attack on the cca2-secure hqc kem. In *International Conference on Smart Card Research and Advanced Applications*, pages 119–134. Springer, 2020.