POLITECNICO DI TORINO

Master's Degree in Computer Engineering

Politecnico di Torino

Master's Degree Thesis

Analysis and Contributions to a Post-Quantum Cryptography Library written in Rust for a ARM Cortex-M4 board

Supervisors

Prof. ANTONIO JOSE' DI SCALA

Dott. Guido BERTONI

Dott. Maria Chiara MOLTENI

Candidate

Francesco MEDINA

April 2024

Abstract

Nowadays, there is a rapid proliferation of IoT systems motivated by the fact that these devices have a large field of applicability. Embedded systems are often used in IoT devices in order to process, collect, exchange data over the Internet and intercommunicate. These devices introduce new security challenges due to their hardware limitations. At the same time, the research into quantum computing is growing and becoming a tangible reality for the coming decades. As the main consequence, the world is starting to prepare to deal, in terms of cybersecurity, with quantum computers by introducing Post Quantum Cryptography (PQC) which is the set of cryptographic algorithms and protocols designed to allow systems to remain secure even in the presence of quantum computers. This work provides a comprehensive overview of the fundamental mathematical aspects of PQC and presents an efficient and practical solution for ARM Cortex M4 microcontrollers which are widely used for embedded systems and IoT applications for their known performance, versatility and efficiency. In particular, this study is focused on the CRYSTALS-KYBER KEM. In this study, a specific Kyber library written in Rust is selected, and the analysis pays close attention to aspects related to secure programming. For this purpose, we chose Rust system programming language for the programming part for its emphasis on safety at design phase in order to enhance the reliability and security. The objective of this study is to analyze these two technologies and determine their feasibility and efficiency when integrated with ARM Cortex M4 microcontrollers in order to make it secure in the face of a quantum threat. Finally, this work brings improvements for the library and contributions in terms of quality and code maturity.

Summary

By 2023, Internet of Things (IoT) connections will represent 14.8 billion which means nearly half of all global connected devices and connections, according to Cisco Annual Internet Report (2018-2023) [1]. IoT devices have a wide range of applicability in industry and for personal use. These devices introduce new security challenges due to their limitations in terms of energy consumption and computational resources. Embedded systems are often used in IoT industry and unlike ordinary computers they require greater focus on error prevention aspects. In parallel, research into quantum computing is increasing and becoming a tangible reality for the coming decade. Quantum computers will introduce potential risks to our traditional cryptographic systems as demonstrated in 1994 by the Shor's quantum algorithm. Shor's algorithm efficiently solves the problem of integer factorization. Hence, once this algorithm will be consistently executed by a quantum computer, it will break the security of widely-used cryptographic schemes which rely on the complexity of integer factorization problems for their security (i.e., RSA).

As the main consequence, the world is starting to prepare to deal, in terms of cybersecurity, with quantum computers by introducing the Post Quantum Cryptography (PQC) which is the set of cryptographic algorithms and protocols designed to allow systems to remain secure even in the presence of quantum threats.

Given this context, we established some requirements: on one hand, a solution to mitigate weaknesses that may arise when developing with embedded systems, and on the other hand, a solution to integrate a PQC algorithm into embedded devices to maintain confidentiality, integrity, and authenticity against both "classical" and quantum attacks (as depicted in figure 1).

For this case study, we chose to use a PQC library written in Rust which is a system programming language that pays close attention on safety at design phase. Rust enhances reliability and security on embedded systems programming. In particular, this language introduces its own ownership model in order to mitigate common weaknesses such as memory leaks, null pointer dereferences, and data races. More in detail, we considered the CRYSTALS-KYBER Key Encapsulation Mechanism which is one of the finalists in Round 3 in the NIST PQC Standardization

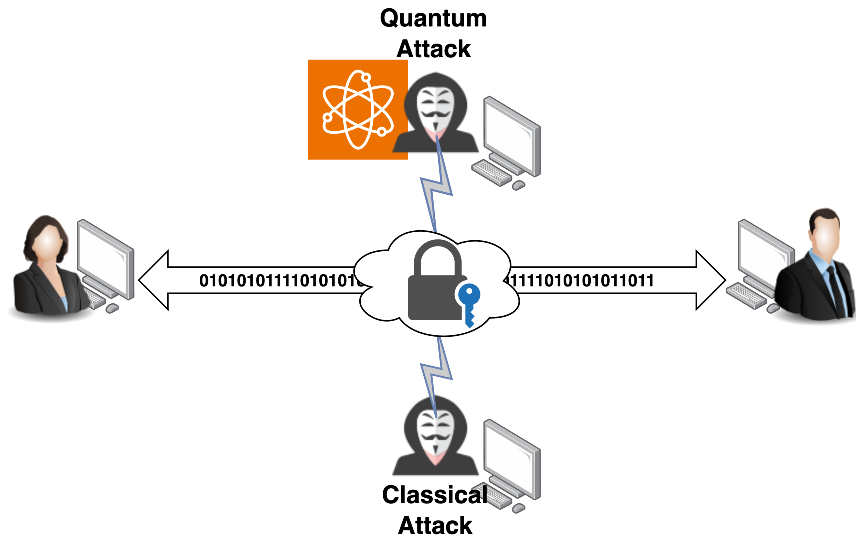


Figure 1: PQC must ensure protection for both Classical or Quantum Attacks methods

process. This scheme can be considered as a great candidate because of its robustness, resilience and interoperability with legacy systems. Kyber's resilience, unlike some traditional cryptographic systems which are vulnerable to Shor's algorithm, derives mainly from difficulty of solving lattice-based mathematical module learning with errors problems, even for quantum computers.

Then, we analyzed the interaction between ARM Cortex M4 boards and the selected library.

During the course of this study, we have devoted particular attention to the Random Number Generation (RNG) module. It can be implemented in software or hardware, and plays a critical role in cryptography, particularly in generating session keys, or cryptographic numbers used once (called nonces). A weak random number generator can compromise the security of any cryptographic protocol (i.e, by using predictive attacks). Since Kyber algorithms used nonces, we observed that the library do not correctly handle potential errors or malfunctions of the RNG peripheral. This scenario represents a non-negligible issue, because IoT devices are specially designed to be intensively used, leading to a higher probability of failure and service disruption.

To address this problem we realized tests and implement software contributions, accepted and released by the library community, which brings robustness, failure recovery, security and better code quality.

Acknowledgements

I would like to express my sincere gratitude to my parents, grandparents, and my brother for their trust and tremendous support throughout this academic journey.

A special thanks to Prof. Antonio José Di Scala for his professionalism, availability, and for the positive experience I perceived while working on this thesis.

I am deeply grateful to Dr. Maria Chiara Molteni for her exquisite patience and expertise, which have supported me greatly during this work.

Lastly, I would like to express my gratitude to Dr. Guido Bertoni for his availability and for providing me with the opportunity to carry out this thesis.

"The measure of intelligence
is the ability to change"
Albert Einstein

"La misura dell'intelligenza è data
dalla capacità di cambiare
quando è necessario"
Albert Einstein

Table of Contents

List of Tables x

List of Figures xi

Acronyms xv

1 Introduction 1
- 1.1 Thesis objectives . . . 2
- 1.2 Thesis structure . . . 2

2 Post-Quantum Cryptography 4
- 2.1 Nowadays Security . . . 4
  - 2.1.1 Security Problems and Constraints . . . 5
- 2.2 Post Quantum Cryptography (PQC) . . . 5
- 2.3 Lattices . . . 6
  - 2.3.1 Preliminaries . . . 6
  - 2.3.2 Definition of Lattice . . . 7
  - 2.3.3 Definition of Basis of the lattice . . . 7
  - 2.3.4 Properties of the Lattice . . . 9
  - 2.3.5 Lattice-based Hard Problems . . . 10
  - 2.3.6 Lattice-based Cryptosystem . . . 13
- 2.4 LWE - Learning With Errors . . . 15
  - 2.4.1 Definition . . . 15
- 2.5 Public-Key Cryptography using LWE . . . 18
  - 2.5.1 Key Generation . . . 18
  - 2.5.2 Encryption . . . 19
  - 2.5.3 Decryption . . . 21
  - 2.5.4 NIST Standardization Competition . . . 23

3 The Number Theoretic Transform (NTT) and its inverse (INTT) 26
- 3.1 NTT . . . 26

3.1.1 Definitions . . . 26
3.1.2 NTT-based Polynomial Multiplication . . . 29
3.1.3 FFT Trick for Negacyclic-based NTT $\in \mathbb{Z}_q(x)/(x^n+1)$ . . . 30

4 Kyber 35
4.1 Preliminaries . . . 35
4.1.1 Oracle . . . 35
4.1.2 Security Goals . . . 36
4.1.3 IND-CPA . . . 36
4.1.4 IND-CCA . . . 37
4.1.5 Implications . . . 37
4.2 CRYSTALS . . . 38
4.2.1 Functions and Parameters Employed . . . 38
4.2.2 Kyber's Polynomials and Coefficients . . . 39
4.3 Security Approach . . . 39
4.4 Kyber.CPAPKE . . . 40
4.4.1 Key Generation . . . 40
4.4.2 Decryption . . . 43
4.5 Kyber.CCAKEM . . . 44
4.5.1 Key Generation . . . 44
4.5.2 Encapsulation (Client → Server) . . . 45
4.5.3 Decapsulation (Server → Client) . . . 45

5 Rust Programming Language 47
5.1 Introduction . . . 47
5.2 Preliminaries . . . 47
5.2.1 Vulnerability . . . 47
5.2.2 Weakness . . . 48
5.3 CVE for Linux Kernel . . . 48
5.4 CWE for C language . . . 49
5.5 Rust Overview . . . 55
5.5.1 Safety - Pointers and Memory . . . 55
5.5.2 Syntax . . . 56
5.5.3 Ownership . . . 57
5.5.4 Borrowing . . . 59
5.5.5 Mutability . . . 59
5.5.6 Lifetimes . . . 60
5.5.7 Polymorphism . . . 62
5.6 The no_std environment . . . 64
5.7 Motivation of adopting Rust for the project . . . 64

6 Kyber library written in Rust on a ARM Cortex-M4 65
6.1 The STM32F303VCT6 Board 65
6.1.1 Development environment for STM32 65
6.1.2 Our selection 65
6.1.3 Limitations of the board 67
6.2 State of Art of Kyber on a ARM Cortex-M4 67
6.2.1 PQClean 67
6.2.2 pqm4 68
6.3 Selected Rust Kyber Library for the board 68
6.4 Performance Evaluation 68
6.4.1 Clock Cycles 68
6.4.2 Clock Cycles Measurement Method 69
6.5 Code Coverage analysis 74
6.5.1 Code Coverage 74
6.5.2 Code Coverage Criteria 74
6.5.3 Test Coverage 75

7 Random Number Generator 77
7.0.1 FIPS 140-2 77
7.1 The STM32F407VGT6 Board 78
7.1.1 TRNG Functional Description 78
7.1.2 TRNG Workflow 78
7.1.3 TRNG Error Management 79
7.2 Randomness Tests 81
7.2.1 Test Description 81
7.2.2 Results 81

8 Contributions to the library 83
8.1 Performance Tests for STM32F4 Board 83
8.1.1 Results for STM32F4 84
8.2 Performance Results for STM32F3 board 88
8.3 Code Coverage Results 92
8.4 Contributions to the library 93
8.4.1 Solution for the Unhandled RNG exceptions 93
8.4.2 Solution advantages 94
8.4.3 Contributions to enhance Code Quality 98

9 Conclusions 103

A Boundary Value Coverage Observations 104

Bibliography 107

List of Tables

2.1 In the first round the total lattice-based candidates account for 26 out of 64. Candidates "HILA5" and "Round2" were withdrawn and merged in "Round5" candidate. 24
2.2 Second round was announced on January 30, 2019. This time the lattice-based candidates accounted for 12 out of 26 24
2.3 Third round announced on July 22 2020, NIST included two new candidates (*) for the lattice-based type to be considered in the fourth round, they were "FrodoKEM" and "NTRU Prime". So now lattice-based accounted 7 out of 15 25

4.1 Kyber Parameters 39
4.2 Kyber Functions 39
4.3 Secret and Public key size depend on chosen Kyber's version 40

5.1 Overflow and Memory Corruption vulnerabilities for the year 2022 . 50
5.2 Weaknesses in Software Written in C, submission date 2008-04-11 [24]. Some of them are highlighted in gray and are prevented at compile-time by the Rust compiler 53

8.1 Speed comparison at 24MHz for the STM32F407VGT6 board based on the language and method used. The average, minimum and maximum value of each algorithms are reported for each KEM method 84
8.2 Speed comparison at 24MHz based on the language and method used. The average, minimum and maximum value of each algorithms are reported for each KEM method 88

List of Figures

1 PQC must ensure protection for both Classical or Quantum Attacks methods . . . iii

2.1 Basis represented by the vectors $b1$ and $b2$ . . . 8
2.2 Integer linear combination of the basis with the vector s, generates the point t . . . 9
2.3 Vectors $\mathbf{b_1}$ and $\mathbf{b_2}$ are the 2 linearly independent vectors contained in the ball with radius λ_2 (successive minima) centered at the origin. 10
2.4 In this Decision variant (SVP), vector $\mathbf{v} \in \mathcal{L}$ is the shortest vector to origin, in the lattice $\mathcal{L}$ generated by basis b_1 and b_2 . . . 11
2.5 Vector $\mathbf{v} \in \mathcal{L}$ is the closest to vector $\mathbf{w} \in \mathbb{Q}^n$ in the lattice $\mathcal{L}$ generated by basis b_1 and b_2 . . . 13
2.6 Alice's secret key on the left. Alice's public key on the right . . . 14
2.7 Key Generation Function (computed by Alice) where k corresponds to the rank d and $m = n$ . . . 19
2.8 Encryption Function (computed by Bob) . . . 20
2.9 Formula to compute v (computed by Bob) . . . 21
2.10 Formula to compute u (computed by Bob) . . . 21
2.11 Rounding according to Ding's technique. Probability of failure is 2^{-10} 22
2.12 Decryption Function (computed by Alice) . . . 23

3.1 Binary Tree representing the Chinese Remainder Theorem map of FFT Trick over $\mathbb{Z}_q[x]/(x^n + 1)$ . . . 31
3.2 Cooley-Tukey (CT) butterfly . . . 32
3.3 Gentleman-Sande (GS) butterfly . . . 33

4.1 Example of an Encryption Oracle . . . 35
4.2 Kyber.CPAPKE.KeyGen() [18] . . . 41
4.3 Pseudocode of the Parse [18] function . . . 41
4.4 Pseudocode of the centered binomial distribution (CBD) [18] . . . 42
4.5 Kyber.CPAPKE.Enc(pk, m, r) [18] . . . 43
4.6 Kyber.CPAPKE.Dec(sk, c) [18] . . . 44

4.7 Fujisaki-Okamoto Transform (FO) for IND-CCA2-secure Decapsulation 44
4.8 Kyber.CCAKEM.KeyGen() [18] 45
4.9 Kyber.CCAKEM.Enc(pk) [18] 45
4.10 Kyber.CCAKEM.Dec(c, sk) [18] 46

5.1 The list of all CVE recorded from 1999 to 2023. [23] 49

6.1 STM32f303VCT6 Layout (top view) [28] 66
6.2 List of known flaws. On Each flaw it is specified which tests might have detected them, from [30] 67

7.1 Front view of the STM32F407VGT6 board from [35] 79
7.2 RNG Block Diagram from [36] 80
7.3 RNG Schema from [37] 80
7.4 Bar chart depicting the probability distribution of each integer from 0 to 32000 considering 40 billion draws 82
7.5 Enlarged surface of the previous graph 82

8.1 Kyber-512 KEM functions measured on STM32F407VGT6 board using DWT method 85
8.2 Kyber-768 KEM functions measured on STM32F407VGT6 board using DWT method 85
8.3 Kyber-1024 KEM functions measured on STM32F407VGT6 board using DWT method 86
8.4 Kyber-512 KEM functions measured on STM32F407VGT6 board using SysTick method 86
8.5 Kyber-768 KEM functions measured on STM32F407VGT6 board using SysTick method 87
8.6 Kyber-1024 KEM functions measured on STM32F407VGT6 board using SysTick method 87
8.7 Kyber-512 KEM functions measured on STM32F303VCT6 board using SysTick approach 89
8.8 Kyber-768 KEM functions measured on STM32F303VCT6 board using SysTick approach 89
8.9 Kyber-1024 KEM functions measured on STM32F303VCT6 board using SysTick approach 90
8.10 Kyber-512 KEM functions measured on STM32F303VCT6 board using DWT approach 90
8.11 Kyber-768 KEM functions measured on STM32F303VCT6 board using SysTick approach 91
8.12 Kyber-1024 KEM functions measured on STM32F303VCT6 board using SysTick approach 91

8.13 Coverage results for Kyber project root folder 92
8.14 Coverage results for src folder 92
8.15 Coverage results for src/reference folder 93
8.16 Added a new error enum called RandomBytesGenerator 94
8.17 Modified the *randombytes()* function in order to use the try_fill_bytes() method and handle the error case 94
8.18 IND-CCA Key Generation RNG issue 95
8.19 Our IND-CCA Key Generation RNG solution 96
8.20 IND-CCA Encapsulation RNG issue 97
8.21 Our IND-CCA Encapsulation RNG solution 98
8.22 Merged pull request `https://github.com/Argyle-Software/kyber/commit/5e931d6d8ca536a98308c2faa0b505f3e2c9949b` 99
8.23 Merged pull request `https://github.com/Argyle-Software/kyber/commit/5e931d6d8ca536a98308c2faa0b505f3e2c9949b` 100
8.24 Merged pull request `https://github.com/Argyle-Software/kyber/commit/5e931d6d8ca536a98308c2faa0b505f3e2c9949b` 100

Acronyms

DLP
: Discrete Logarithm Problem

ECDLP
: Elliptic-curve Discrete Logarithm Problem

LWE
: Learning With Errors

M-LWE
: Module-based Learning With Errors

NIST
: National Institute of Standards and Technology

PQC
: Post Quantum Cryptography

CWE
: Common Weakness Enumeration

CVE
: Common Vulnerabilities and Exposures

NTT
: Number Theoretic Transform

INTT
: Inverse Number Theoretic Transform

FIPS
: Federal Information Processing Standards Publication

IND-CPA
: Indistinguishability under Chosen Plaintext Attack

IND-CCA1
: Indistinguishability under non-adaptive chosen ciphertext attack

IND-CCA2
: Indistinguishability under adaptive chosen ciphertext attack

Chapter 1

Introduction

The proverbial phrase *"Prevention is better than cure"* (cit. Desiderius Erasmus ~1500) means it is preferable and relatively simple to stop problems in the first place rather than repairing the damage after it occurs. This proverb, often applied in healthcare, emphasizes the importance of taking preventive measures to avoid potential issues, rather than realizing solutions to resolve the problems once they arise.

Similarly, in a cybersecurity context, prevention is highly important because taking proactive measures can help mitigate risks and prevent negative outcomes. Nowadays, the principle of prioritizing security above all else is well-known. Security by design refers to the integration of a concrete solution designed and implemented from the initial stages of a computer product's lifecycle.

In this work we adopted this principle by selecting two technologies specifically designed to provide preventive mechanisms against future attacks, damages, errors, and other types of issues. These two elements are:

- **Post Quantum Cryptography**: which is the set of cryptographic algorithms and protocols designed to allow systems to remain secure even in the presence of quantum threats. In this context, we take in consideration the **CRYSTALS-KYBER** KEM for our analysis.

- **Rust Embedded**: this system programming language can help developers to mainly focus on aspects regarding safety. IoT devices introduce new security challenges due to their hardware limitations. Embedded systems, unlike commercial computers, are very raw in terms of management and require a greater focus on prevention aspects, which Rust natively offers.

We merged these two elements by integrating them on **ARM-Cortex M4** boards and observing how they behave providing some contributions.

1.1 Thesis objectives

Overall, the objective of this study is to analyze and determine the feasibility and efficiency of a Kyber library written in Rust when integrated with ARM Cortex M4 microcontrollers. In parallel, bring improvements for the library and contributions in terms of code quality.

1.2 Thesis structure

- Chapter 2: provides the mathematical background necessary to understand the Kyber schema. Section 2.3 presents the definition, properties and problems of Lattices. Section 2.4 describes the Learning With Errors and its variants. Section 2.5 provides an example of Public-Key Cryptosystem using LWE and Results of the NIST Standardization Competition

- Chapter 3: is a theoretical overview of the Number Theoretic Transform (NTT) and its Inverse (INTT)

- Chapter 4: presents Kyber. This chapter starts providing some preliminaries about current security goals. Section 4.3 explains the Kyber's security approach which employs a dual-phase methodology described in section 4.4 and 4.5.

- Chapter 5: is the introduction to the Rust language. This chapter starts explaining the importance of being aware of the impacts generated by weaknesses within a software, by collecting in section 5.3 and 5.4 some relevant real cases and statistics of CVEs and CWEs. Then, the overview of the language is presented in section 5.5 and 5.6. Section 5.7 motivates the use of the language in our work.

- Chapter 6: presents the state of the art of Kyber on the ARM Cortex-M4 microcontrollers and the selected Kyber library for this work (section 6.1 6.2 and 6.3). Section 6.4 and 6.5 provide an explanation of sampling and measurement techniques used to perform the speed tests and coverage evaluation.

- Chapter 7: briefly describe the relevance of a Random Number Generator. Section 7.1 presents the selected board for such analysis, and section 7.2 provides the corresponding randomness Tests realized with an explanation of results as well.

- Chapter 8: is the presentation of findings of the computed performance tests, including the interpretation of results (section 8.1 and 8.2). Section 8.3,

explains the code coverage results computed. Contributions to the library are presented in section 8.4.

- Chapter 9: contains the final considerations and the summary of key findings and recommendations for future research about this thesis work.

Chapter 2

Post-Quantum Cryptography

2.1 Nowadays Security

Presently, we feel secure knowing that our encrypted data is safe because in order to decrypt the information, most of the time it would take the attacker a considerable amount of time. One of the important goal of security mechanisms and protocols, is to establish a secure channel of communication. The solutions adopted in standardized technologies take advantage of well-known mathematical problems that have a high complexity of resolution for the nowadays technology knowledge, they are:

1. Integer Factorization Problem
2. Discrete Logarithm Problem (DLP)
3. Elliptic-curve Discrete Logarithm Problem (ECDLP)

Algorithms are divided according to the type of function they perform within a security mechanism and whether they are symmetrical or asymmetrical:

Symmetric:

- Hash function (e.g. SHA-2, SHA-3)
- Block or stream cipher (e.g. AES, ChaCha20)
- Authenticator (e.g. HMAC, GMAC, Poly1305)

Asymmetric:

- Public-key Encryption (PKE) (e.g. RSA, Diffie-Hellman, ECDH)
- Signatures (e.g. RSA, DSA, ECDSA)

2.1.1 Security Problems and Constraints

As time passes, the phenomenon of quantum computers takes hold. In 1994 the American mathematician Peter W. **Shor** demonstrates [2] that if a quantum computer uses Shor's algorithm in order to find the prime factors of an integer, it can be able to efficiently solve this problem in a polynomial time. Therefore, this algorithm is able to break schemes based on the concept that factoring large integers is computationally hard. The victims will be the previous mention PKE schemes like RSA (that will be insecure by 2030 [3]), Diffie-Hellman, ECDH. More in detail, the complexity class of this decision problem is called bounded-error quantum polynomial time (BQP) because it can be solved in polynomial time with an error probability $\leq \frac{1}{3}$ for all instances [4].

Therefore, is very important to prevent these future attacks through mechanisms that not only guarantees security in a quantum era but also being interoperable with the current infrastructures present all over the world and being compatible with most of the current technologies considering not only ordinary computers but also embedded systems that has less resources in terms of energy and memory space.

2.2 Post Quantum Cryptography (PQC)

Post Quantum Cryptography (PQC) is the set of cryptographic algorithms designed with the purpose of being able to preserve security in a context of quantum supremacy which means the era where quantum computers will be a reality in the world and will be capable of breaking the security of current cryptographic standards by making them no longer secure. More precisely, PQC is an asymmetric cryptography that is not based on factoring or discrete logarithm problems and is able to resist classical and quantum computers. The main requirement is to ensure integration of PQC into existing security protocols, such as TLS for secure communication over the internet and maintain confidentiality, integrity, and authenticity against both "classical" or quantum attacks.

Among all available families of post-quantum cryptography, only the lattice-based is the most promising one due to its performance, communication bandwidth and efficiency compared with the other schemes. Another important point to consider is that the most common cryptographic primitives (PKE, KEM, digital signature, key exchange, etc.) can be implemented using lattices (explained on the next paragraphs).

For the purposes of studying CRYSTALS-KYBER we focus on the study of the lattice-based approach. Some post-quantum cryptographic schemes:

1. **the lattice-based**
2. the isogeny-based
3. the hash-based
4. the code-based
5. the multivariable-based
6. the rank-based

2.3 Lattices

Since 18th century lattices have been used in mathematics. The use of lattices in the design of cryptographic schemes started in the late '90s.

2.3.1 Preliminaries

Subgroup Let consider the group $\mathbb{R}^n$, a nonempty subset $\mathcal{S} \subseteq \mathbb{R}^n$ is called (additive) subgroup if both conditions are satisfied:

$$\mathbf{v}, \mathbf{w} \in \mathcal{S} \Rightarrow \mathbf{v} + \mathbf{w} \in \mathcal{S}$$
$$\mathbf{v} \in \mathcal{S} \Rightarrow -\mathbf{v} \in \mathcal{S}$$

In other words, the subgroup $\mathcal{S}$ is closed under addition, so the sum (or subtraction) between two points of the subgroup $\mathcal{S}$ generates another point inside $\mathcal{S}$.

The previous two conditions implies the existence of the identity element $\mathbf{0} \in \mathcal{S}$ because $\mathbf{v} + (-\mathbf{v}) = \mathbf{0} \in \mathcal{S}$.

Discrete subgroup Let define the subgroup $\mathcal{S}$, a subgroup is called discrete if exists a constant $\varepsilon > 0$ such that

$$\forall \mathbf{v} \in \mathcal{S} \mid \mathbf{v} \neq 0$$

$$\|\mathbf{v}\| \geq \varepsilon$$

or an equivalent definition, if any two distinct points of the subgroup $\mathbf{v} \neq \mathbf{w} \in \mathcal{S}$ are at least spaced out

$$\|\mathbf{x} - \mathbf{y}\| \geq \varepsilon$$

.

Unimodular matrix An integer matrix $\mathbf{U} \in \mathbb{Z}^{n \times n}$ is called unimodular matrix if

$$\det(\mathbf{U}) = \pm 1$$

2.3.2 Definition of Lattice

An n-dimensional lattice $\mathcal{L}$ is defined as a subset of $\mathbb{R}^n$ that satisfy the condition of being a discrete additive subgroup.

Geometrically speaking a lattice is an n-dimensional space populated by points with a periodic pattern.

2.3.3 Definition of Basis of the lattice

A basis $\mathbf{B} = \{\mathbf{b}_1, \ldots, \mathbf{b}_n\} \subset \mathbb{R}^n$ of a lattice $\mathcal{L}$ is a set of linearly independent vectors whose integer linear combinations generate the lattice:

$$\mathcal{L} = \mathcal{L}(\mathbf{B}) := \left\{ \sum_{i=1}^{n} z_i \mathbf{b}_i : z_i \in \mathbb{Z} \right\}$$

If we denote $\mathbf{B} \in \mathbb{R}^{n \times n}$ a nonsingular matrix whose (ordered) columns are $\mathbf{b}_1, \ldots, \mathbf{b}_n$, then another equivalent way to describe a lattice can be:

$$\mathcal{L} = \mathbf{B} \cdot \mathbb{Z}^n = \{\mathbf{Bz} : \mathbf{z} \in \mathbb{Z}^n\}$$

The above definition describe an equivalent definition in which a lattice $\mathcal{L}$ can be obtained by applying some nonsingular linear transformation to the integer lattice $\mathbb{Z}^n$.

Two bases $\mathbf{B}_1, \mathbf{B}_2$ generate the same lattice $\mathcal{L}$ if and only if there exists a unimodular $\mathbf{U} \in \mathbb{Z}^{n \times n}$ such that $\mathbf{B}_1 = \mathbf{B}_2\mathbf{U}$

Proof Let suppose $\mathcal{L}(\mathbf{B}_1) = \mathcal{L}(\mathbf{B}_2)$, then each column of $\mathbf{B}_1$ is an integer combination of the columns of $\mathbf{B}_2$ and vice-versa. That means, exist $\mathbf{U}, \mathbf{V} \in \mathbb{Z}^{n \times n}$ such that $\mathbf{B}_1 = \mathbf{B}_2\mathbf{U}$ and $\mathbf{B}_2 = \mathbf{B}_1\mathbf{V} = \mathbf{B}_2\mathbf{UV}$ and since $\mathbf{B}_2$ is nonsingular (invertible) it holds

$$\mathbf{B}_2^{-1}\mathbf{B}_2 = \mathbf{B}_2^{-1}\mathbf{B}_2\mathbf{UV}$$
$$\mathbf{I} = \mathbf{UV}$$

where $\mathbf{I}$ is the identity matrix and must occur that $\det(\mathbf{U}) \cdot \det(\mathbf{V}) = 1$, therefore both $\det(\mathbf{U}) = \det(\mathbf{V}) = 1$ or $\det(\mathbf{U}) = \det(\mathbf{V}) = -1$

In conclusion, **the same lattice can be generated by different basis (if $\mathbf{B}_i^{-1} \cdot \mathbf{B}_j$ is unimodular) and not by a unique one. It is precisely on this intrinsic mathematical feature that makes lattice-based cryptography possible.**

Example Let define the lattice $\mathcal{L}$ generated by the following basis points represented by matrix A and depicted in figure 2.1

$$A = \begin{pmatrix} 1 & 1 \\ 2 & -1 \end{pmatrix} \subseteq \mathbb{Z}^2$$

.

Let $t \in \mathcal{L}$ be the point generated by the linear combination with the vector $s = [2,1] \in \mathbb{Z}^2$ depicted by figure 2.2

$$t = A \cdot s = \begin{pmatrix} 1 & 1 \\ 2 & -1 \end{pmatrix} \cdot \begin{pmatrix} 2 \\ 1 \end{pmatrix} = \begin{pmatrix} 3 \\ 3 \end{pmatrix}$$

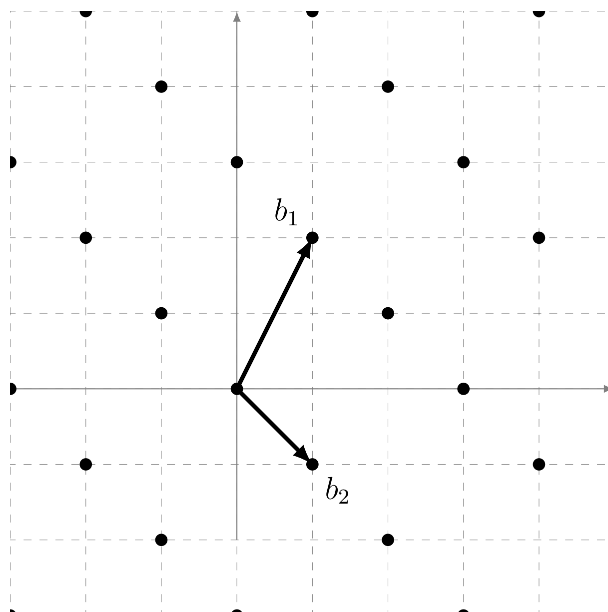


Figure 2.1: Basis represented by the vectors $b1$ and $b2$

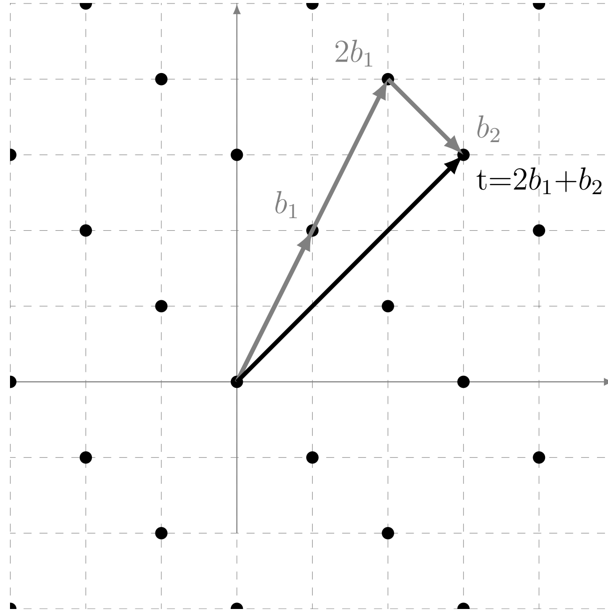


Figure 2.2: Integer linear combination of the basis with the vector s, generates the point t

2.3.4 Properties of the Lattice

Rank

The rank k of a lattice $\mathcal{L} \subset \mathbb{R}^n$ is the dimension of its linear span.

$$k = \dim(\text{span}(\mathcal{L}))$$

When $k = n$, the lattice is said to be full-rank.

Volume

Let denote $\mathbf{B}$ as the set of basis of $\mathcal{L}$. The volume of a lattice $\mathcal{L}$ is defined as:

$$vol = \det(\mathcal{L}) = |\det(\mathbf{B})|$$

Minimum distance

The minimum distance of $\mathcal{L}$ is the smallest distance between any two lattice points. Can be also interpreted as the length of the shortest nonzero lattice vector $\mathbf{v} \in \mathcal{L}$ (minimum length under the Euclidean norm):

$$\lambda(\mathcal{L}) := \min_{\mathbf{v} \in \mathcal{L} \setminus \{0\}} \|\mathbf{v}\| = \min_{\text{distinct } \mathbf{x},\mathbf{y} \in \mathcal{L}} \|\mathbf{x} - \mathbf{y}\|$$

Successive minima

Let $n \in \mathbb{N}$ be the lattice dimension and let $i \in \mathbb{N}$ such that $i \leq n$ and $\lambda_i(\mathcal{L})$ be the smallest radius such that the closed ball $\mathcal{B}(0, r)$ of radius $r > 0$ centered at the origin contains at least i linearly independent vectors of length at most r. The definition of successive minima generalize the previous minimum distance concept in which $\lambda = \lambda_1$ so that the following sequence of parameters are called the successive minima of the lattice $\mathcal{L}$:

$$\lambda_1 \leq \lambda_2 \leq ... \leq \lambda_n$$

Example of Successive Minima The successive minima λ_2 is the radius of the smallest ball containing 2 linearly independent lattice vectors ($\mathbf{b_1}$ and $\mathbf{b_2}$) as depicted in figure 2.3.

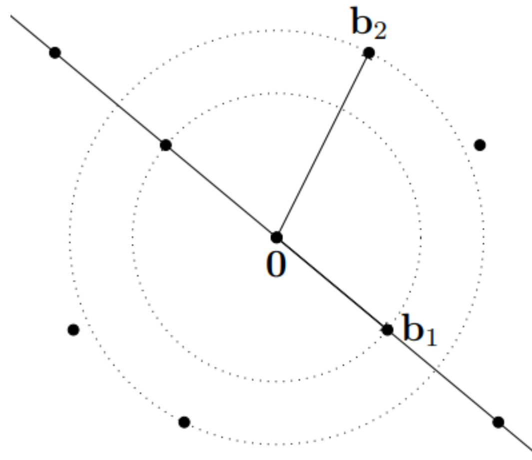


Figure 2.3: Vectors $\mathbf{b_1}$ and $\mathbf{b_2}$ are the 2 linearly independent vectors contained in the ball with radius λ_2 (successive minima) centered at the origin.

2.3.5 Lattice-based Hard Problems

Using the concept of minimum distance and successive minima, several NP-hard problems have been studied. The most relevant problems are discussed in the next paragraphs.

SVP - Shortest Vector Problem

There are two versions of the shortest vector problem and each of these versions contains 3 variants.

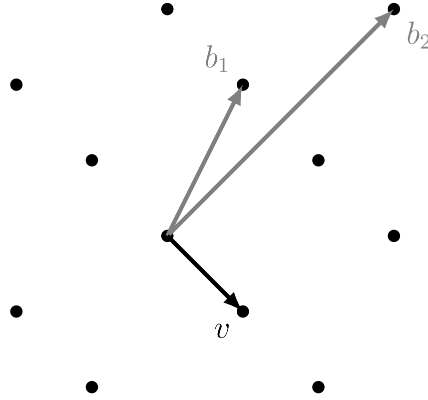


Figure 2.4: In this Decision variant (SVP), vector $\mathbf{v} \in \mathcal{L}$ is the shortest vector to origin, in the lattice $\mathcal{L}$ generated by basis b_1 and b_2

SVP - Exact Form

Let denote with $\mathbf{B}$ the lattice basis and $d \in \mathbb{R}^n \mid d > 0$. There are the 3 variants:

- **Decision** the goal of this variant is to differentiate the following two cases:

$$\lambda_1(\mathcal{L}(\mathbf{B})) \leq d$$

$$\lambda_1(\mathcal{L}(\mathbf{B})) > d$$

- **Calculation** the purpose of this version is to find the minimum distance

$$\lambda_1(\mathcal{L}(\mathbf{B}))$$

- **Search** the goal is to find a nonzero $\mathbf{v} \in \mathcal{L}(\mathbf{B})$ such that:

$$\|\mathbf{v}\| = \lambda_1(\mathcal{L}(\mathbf{B}))$$

Figure 2.4 depicts this problem.

SVP_γ - Approximate Form

Let denote with $\mathbf{B}$ the lattice basis, $d \in \mathbb{R}^n \mid d > 0$ and the approximation factor $\gamma = \gamma(n) \geq 1$ that is a function of the dimension n. These are the 3 variants:

- **Decision** ($GapSVP_\gamma$) distinguish between the cases

$$\lambda_1(\mathcal{L}(\mathbf{B})) \leq d$$

$$\lambda_1(\mathcal{L}(\mathbf{B})) > \gamma \cdot d$$

- **Estimation ($EstSVP_\gamma$)** compute $\lambda_1(\mathcal{L}(\mathbf{B}))$ up to a γ factor, i.e., output some $d \in [\lambda_1(\mathcal{L}(\mathbf{B})), \gamma \cdot \lambda_1(\mathcal{L}(\mathbf{B}))]$

- **Search (SVP_γ)** search a (nonzero) $\mathbf{v} \in \mathcal{L}(\mathbf{B})$ such that

$$0 < \|\mathbf{v}\| \leq \gamma \cdot \lambda_1(\mathcal{L}(\mathbf{B}))$$

Considering $\gamma = 1$ all variants corresponds to the previous described Exact Form problem variants. The problems can become easier as γ increases. Formally, the hardness of $GapSVP_{\gamma'} \leq GapSVP_\gamma$ for any $\gamma' \geq \gamma$. In general, being able to solve SVP$_\gamma$ implies being able to solve EstSVP$_\gamma$, which implies being able to solve GapSVP$_\gamma$ as indicated as follow:

$$\text{GapSVP}_\gamma \leq \text{EstSVP}_\gamma \leq \text{SVP}_\gamma$$

SIVP - Shortest Independent Vectors Problem

Using the concept of successive minima is possible to generalize the task without necessarily finding the shortest vector in the lattice as in SVP. Let denote with $\mathbf{B}$ the lattice basis. All variants are in the Approximate Form.

- **Shortest Independent Vectors Problem ($SIVP_\gamma$)** The goal in this problem is to find linearly independent vectors $\mathbf{v_1}, ..., \mathbf{v_n} \in \mathcal{L}$ such that

$$max_i\|\mathbf{v_i}\| \leq \gamma \cdot \lambda_n(\mathcal{L}(\mathbf{B}))$$

- **Decision ($GapSIVP_\gamma$)** Let be d a positive real, the purpose of this variant is to determine if

$$\lambda_n(\mathcal{L}(\mathbf{B})) \leq d \text{ or } \lambda_n(\mathcal{L}(\mathbf{B})) > \gamma \cdot d$$

- **Successive Minima Problem (SMP)** search linearly independent vectors $\mathbf{v_i}, ..., \mathbf{v_n} \in \mathcal{L}$ such that

$$\|\mathbf{v_i}\| \leq \gamma \cdot \lambda_i(\mathcal{L}(\mathbf{B}))$$

  for all i

CVP - Closest Vector Problem

The closest vector problem is a generalization of the previous shortest vector problem. Let define $\mathbf{w} \in \mathbb{Q}^n$ therefore an arbitrary point **not** within the lattice $\mathcal{L}$. The CVP problem consists in finding a vector $\mathbf{v} \in \mathcal{L}$ that minimizes the distance $\|\mathbf{w} - \mathbf{v}\|$.

2.3.6 Lattice-based Cryptosystem

On December 1996 Oded Goldreich, Shafi Goldwasser and Shai Halevi published the GGH [5] encryption cryptosystem based on lattices, in particular it exploits the hard problem concept of the closest vector problem (CVP) and uses the concept of the lattice reduction as a trapdoor one-way function that means a function which is easy to compute in one direction and difficult to compute in the opposite direction without knowing additional information called the "trapdoor".

Lattice reduction as Trapdoor one-way function

Let consider basis almost parallel called "bad basis" and other basis called "good basis" that are vectors nearly orthogonal as depicted in figure 2.6. Giving as an input only bad basis, the goal of the lattice reduction is to find a basis with short and close to be perpendicular vectors, in other words reduce the bad basis into a good basis.

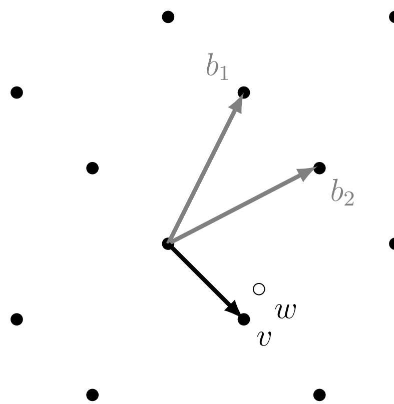
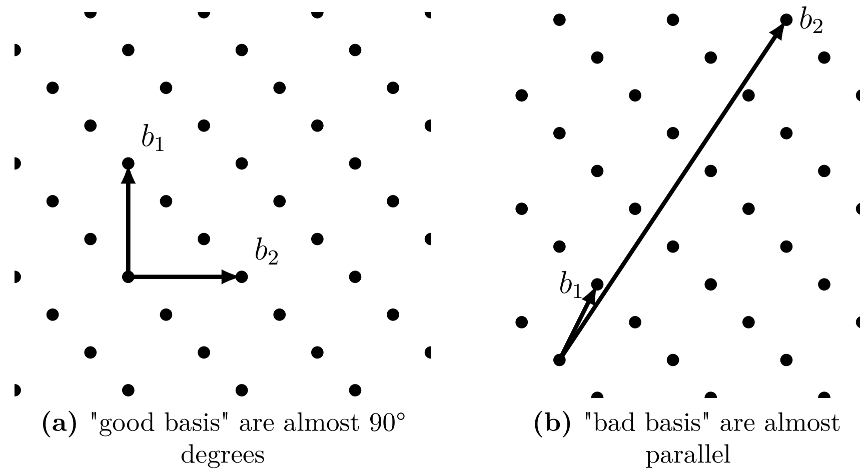


Figure 2.5: Vector $\mathbf{v} \in \mathcal{L}$ is the closest to vector $\mathbf{w} \in \mathbb{Q}^n$ in the lattice $\mathcal{L}$ generated by basis b_1 and b_2



(a) "good basis" are almost 90° degrees

(b) "bad basis" are almost parallel

Figure 2.6: Alice's secret key on the left. Alice's public key on the right

The implementation of this concept can be used for define a trapdoor one-way function. In order to explain the GGH PKE system, let consider the classic informal example of a person (Bob) who wants to send a message to another person (Alice). They have to execute the following steps:

1. Alice sets up a lattice with good basis and bad basis that both generates the same lattice. She uses good basis as a secret and shares her bad basis to Bob as a public key. Since an eavesdropper needs to find the closes lattice point (CVP) using only the bad basis, it's much harder to solve because finding the closest lattice point may come from trying to add and subtract many many vectors.

2. Bob uses Alice's bad basis to embed the message he wants to send. Bob picks a point on the lattice which represents his message and then add a noise. Therefore the new point is slightly moving away from the original point (positioned outside the lattice domain). He sends to Alice the coordinates of this new lattice point.

3. Alice can use her good basis to find the closest lattice point in order to recover the Bob's message.

This encryption was broken in 1999 [6], therefore it is necessary to find something even safer.

2.4 LWE - Learning With Errors

In 2005 the theoretical computer scientist Oded Regev introduced [7] the Learning With Errors (LWE) problem which the task, informally speaking is to learn a secret n-dimensional vector $\mathbf{s} \in \mathbb{Z}_q^n$ given random independent samples of polynomial integers containing a small noise. This problem has been proved [7] to be as hard to solve as several worst-case lattice problems such as GapSVP (the decision version of the shortest vector problem) and SIVP (the shortest independent vectors problem) already discussed. Unlike the GGH, the LWE problem will be used to be able to encrypt one bit at a time. Given its great importance 13 years later Regev won the prestigious Gödel Prize.

2.4.1 Definition

Let define the natural positive integers $n > 1, q \geq 2, m \geq n$ where q is prime and corresponds to the modulo, m is the number of equations and n is the number of variables that corresponds to the ring dimension since we will consider only the case $\mathbb{Z}_q^n \approx \mathbb{Z}_q[x]/\langle x^n + 1\rangle$. Let assign the module rank with d a positive integer and with $U()$ the uniform distribution over the set passed as a input parameter of this function. Let $\chi_{\mathbb{Z},q,\alpha}()$ be the discrete Gaussian distribution on $\mathbb{Z}$ of center q with parameter $\alpha \in]0,1[$. We will consider only the case of a discrete LWE distribution. Formally, the discrete LWE distribution $\mathbf{A}_{s,\chi}$ on $\mathbb{Z}_q^n \times \mathbb{Z}_q$ is the probability distribution obtained by choosing:

- $\mathbf{A} \leftarrow U(\mathbb{Z}_q^{m\times n})$
- $\mathbf{s} \leftarrow U(\mathbb{Z}_q^n)$
- $\mathbf{e} \leftarrow \chi_{\mathbb{Z}^m,\alpha q}$ small compared to q

and outputting

$$(\mathbf{A}, \mathbf{b} = \mathbf{A}\mathbf{s} + \mathbf{e} \mod q)$$

Search LWE: the problem of finding $\mathbf{s}$, given $(\mathbf{A}, \mathbf{b} = \mathbf{A}\mathbf{s} + \mathbf{e} \mod q)$

Decision LWE: the problem of distinguishing from $(\mathbf{A}, \mathbf{b})$ with $\mathbf{b}$ uniform.

$$\begin{pmatrix} a_{0,0} & a_{0,1} & \cdots & a_{0,n-1} \\ a_{1,0} & & & \\ \vdots & & & \vdots \\ a_{m-1,0} & \cdots & \cdots & a_{m-1,n-1} \end{pmatrix} \cdot \begin{pmatrix} s_0 \\ s_1 \\ \vdots \\ s_{m-1} \end{pmatrix} + \begin{pmatrix} e_0 \\ e_1 \\ \vdots \\ e_{m-1} \end{pmatrix} = \begin{pmatrix} t_0 \\ t_1 \\ \vdots \\ t_{m-1} \end{pmatrix}$$

Informal LWE Hardness Concept

The hardness proof is formally documented in [7] and informally it's based on the concept of reduction that is an approach that very informally says

"if we can find an algorithm to solve the LWE problem then we have found an algorithm able to solve the Approx GapSVP problem for any instances that we already know is a very hard problem".

If a quantum computer is needed to solve a problem, in order to solve the other problem there is the need to use a quantum computer as well.

LWE over unstructured lattices This case requires a $O(n^2)$ space complexity and $O(n^2)$ of computation complexity.

In order to reduce LWE space complexity (i.e public key size) and improve the performance of the calculations, it was really comfortable to work introducing structured lattices which has many advantages in terms of efficiency and performance, especially an efficient and faster multiplication between integer polynomials bounded by a modulo some quotient ring (i.e $\langle x^n + 1 \rangle$). So that it is necessary to replace $\mathbb{Z}$ set by the ring of integers R of some number field K. The ring R is isomorphic to $\mathbb{Z}^n$. For instance:

$$R = \mathbb{Z}[x] / \langle x^n + 1 \rangle \tag{2.1}$$

$$K = \mathbb{Q}[x] / \langle x^n + 1 \rangle \text{ with } n = 2^\ell \tag{2.2}$$

At this point, after this modification the multiplication of two integers becomes $a \cdot b \in R_q \mod (x^n + 1)$, whereas without structured lattices it was $a \cdot b \in \mathbb{Z}_q$.

Moreover, structured matrices allows to compute polynomials multiplications modulo quotient ring using the rotation operator $rot()$ so that informally

$$a(x) \cdot b(x) \mod (x^n + 1) = rot(a)_{matrix} \cdot b_{vector}$$

The explicit form of $rot(\mathbf{a})$ depends on the particular field K. In the case where K is a cyclotomic power-of-two field, i.e. $K = \mathbb{Q}[x] / \langle x^n + 1 \rangle$ for power-of-two n, we have the following negacyclic matrix:

$$rot(\mathbf{a}) = \begin{pmatrix} a_0 & -a_{n-1} & -a_{s-2} & \dots & \dots & -a_1 \\ a_1 & a_0 & -a_{n-1} & \ddots & \ddots & -a_2 \\ \vdots & \vdots & \ddots & \ddots & \ddots & \vdots \\ \vdots & \vdots & \ddots & \ddots & \ddots & \vdots \\ a_{n-1} & a_{n-2} & \dots & \dots & \dots & a_0 \end{pmatrix} \tag{2.3}$$

Example Considering $n = 4$ and a ring of integers $R = \mathbb{Z}[x]/\langle x^4 + 1\rangle$. Let define $\mathbf{a} = 3x^3 + 7x^2 - 4x + 5$ and $\mathbf{s} = -x^3 - x^2 + 2x + 3$, both $\mathbf{a}, \mathbf{s} \in R$. The result of the multiplication of them can be easily reduced modulo quotient ring considering that $x^4 = -1$, $x^{n+1} = -x$ etc. because n is power of 2 so that $x^4 + 1 = 0$.

$$\begin{aligned}\mathbf{a} \cdot \mathbf{s} &= -3x^6 - 10x^5 + 3x^4 + 22x^3 + 8x^2 - 2x + 15 \\ &= -3(-\mathbf{x^2}) - 10(-\mathbf{x}) + 3(-\mathbf{1}) + 22x^3 + 8x^2 - 2x + 15 \\ &= 22x^3 + 11x^2 + 8x + 12\end{aligned}$$

Using structured matrices is possible to compute the previous equation with an equivalent way that uses the $rot()$ operator in order to reduce modulo quotient ring. In this example the same structure of 2.3 is applied because the field K is a cyclotomic power-of-two field for power-of-two $n = 4$ (where $n = 2^2$):

$$\mathbf{a} \cdot \mathbf{s} = rot(\mathbf{a}) \cdot \mathbf{s} = \begin{pmatrix} 5 & -3 & -7 & 4 \\ -4 & 5 & -3 & -7 \\ 7 & -4 & 5 & -3 \\ 3 & 7 & -4 & 5 \end{pmatrix} \cdot \begin{pmatrix} 3 \\ 2 \\ -1 \\ -1 \end{pmatrix} = \begin{pmatrix} 12 \\ 8 \\ 11 \\ 22 \end{pmatrix}$$

Ring-LWE This LWE variant uses structured lattices where R is a cyclotomic ring which elements are polynomials of degree less than n. Therefore, using a cyclotomic power-of-two field is possible to compute $\mathbf{As}$ using the previous 2.3 $rot(A)$, therefore the space complexity is decreased to $O(n)$ and the computation complexity this time is $O(nlog(n))$ achieved using NTT algorithms to reduce multiplication complexity of $A \cdot s$ because each columns of the first matrix A are dependent each other so they can be simplified with some tricky methods.
In this structure, in order to increase the security level it requires to increase the size of matrix A. In the below equation, for the first matrix we will apply only one rotation because $m = 1$ and the rank $d = 1$, therefore $rot(\mathbf{a_{1,1}}) \in \mathbb{Z}_q^{n \times n}$.

$$\begin{pmatrix} a_{0,0} & -a_{n-1,0} & -a_{n-2,0} & \cdots & a_{1,0} \\ a_{1,0} & a_{0,0} & -a_{n-1,0} & & \\ a_{2,0} & a_{1,0} & a_{0,0} & & \vdots \\ \vdots & & a_{1,0} & & \vdots \\ a_{n-2,0} & & & & \\ a_{n-1,0} & & \vdots & & a_{0,0} \end{pmatrix} \cdot \begin{pmatrix} s_0 \\ s_1 \\ \vdots \\ s_{n-1} \end{pmatrix} + \begin{pmatrix} e_0 \\ e_1 \\ \vdots \\ e_{n-1} \end{pmatrix} = \begin{pmatrix} t_0 \\ t_1 \\ \vdots \\ t_{n-1} \end{pmatrix}$$

Module-LWE LWE over modules has been considered to be a tradeoff between LWE and R-LWE problems. In particular, it is a generalization of the Ring-LWE

problem since working with the special case Module-LWE$_{d=1}$ corresponds to work with Ring-LWE. Module-LWE guarantees a flexible and scalable level of security (based on tuning a single parameter that is the rank d) compared to Ring-LWE and better computation cost of $O(nd)$ compared to the unstructured lattice version of LWE which has a $O(n^2)$ complexity. Module-LWE use this parameters:

- $\mathbf{A} \leftarrow U(R_q^{m\times d})$
- $\mathbf{s} \leftarrow U(R_q^d)$
- $\mathbf{e} \in R^m$ small compared to q

In the bellow equation since the rank is $d > 1$ and m can be bigger than 1, each element of the first matrix can be consider as multiple rotated matrices of Ring-LWE's $\mathbf{A}$ matrix structure in which $rot(\mathbf{a_{i,j}}) \in \mathbb{Z}_q^{n\times n} \quad i = 1{,}2, .., m \quad j = 1{,}2, ..., d$.

$$\begin{pmatrix} rot(\mathbf{a_{1,1}}) & \cdots & rot(\mathbf{a_{1,d}}) \\ \vdots & \ddots & \vdots \\ rot(\mathbf{a_{m,1}}) & \cdots & rot(\mathbf{a_{m,d}}) \end{pmatrix} \cdot \begin{pmatrix} s_1 \\ \vdots \\ s_m \end{pmatrix} + \begin{pmatrix} e_1 \\ \vdots \\ e_m \end{pmatrix} = \begin{pmatrix} t_1 \\ \vdots \\ t_m \end{pmatrix}$$

2.5 Public-Key Cryptography using LWE

2.5.1 Key Generation

The Key Generation is a function that returns a public key and a secret key and can be more formally defined as following.
Let define $\mathbf{s} \overset{\text{uniform}}{\longleftarrow} Z_q^n, \mathbf{A} \overset{\text{uniform}}{\longleftarrow} Z_q^{n,m}, \mathbf{e} \leftarrow \chi^m$

$$\text{KeyGen}\left(\mathbf{1}^n\right) : \left(pk = \left(\mathbf{A}, \mathbf{t} = \mathbf{As} + \mathbf{e}\right), sk = \mathbf{s}\right)$$

- $\mathbf{s}$ represents the **secret key** or sk
- the concatenation of ($\mathbf{A}$,$\mathbf{t}$) is the **public key** or pk

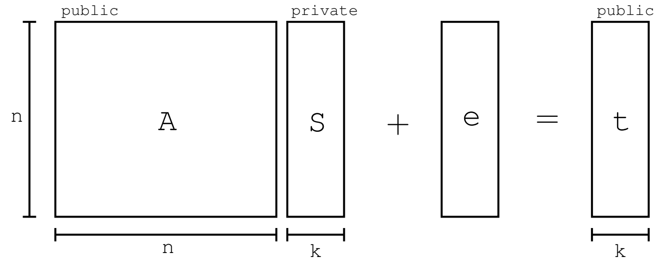


Figure 2.7: Key Generation Function (computed by Alice) where k corresponds to the rank d and $m = n$

Example If $n = 4, q = 89$ and $d = 1$:

$$\begin{pmatrix} 85 & 53 & 10 & 53 \\ 15 & 37 & 24 & 31 \\ 81 & 51 & 3 & 61 \\ 12 & 26 & 29 & 15 \end{pmatrix} \cdot \begin{pmatrix} 1 \\ 0 \\ 1 \\ 0 \end{pmatrix} + \begin{pmatrix} -1 \\ 0 \\ 1 \\ -1 \end{pmatrix} = \begin{pmatrix} 5 \\ 39 \\ 85 \\ 40 \end{pmatrix}$$

2.5.2 Encryption

Once the public-key is shared, the goal of this step is to produce a ciphertext that is the concatenation of v and $\mathbf{u}$ computed with the function below. The parameter μ represents the single bit input message.
Let define $\mathbf{r} \xleftarrow{\text{uniform}} \{0,1\}^m$ and the message bit $\mu \in \{0,1\}$. Finally, let denote with e_1 and e_2 the error polynomial vectors.

$$\text{Enc}_{pk}(\mu) : \left(\mathbf{u} = \mathbf{rA} + \mathbf{e_1}, \text{v} = \mathbf{rt} + \mathbf{e_2} + \mu \left\lfloor \frac{q}{2} \right\rfloor\right)$$

u is in bold because is a vector of polynomials and v is only a polynomial.

Example Let considering to send only one bit (0 or 1) message and continuing with the same values as in the previous example. Notice that the same procedure is repeated if we have to encrypt a buffer of bits such as the letter "C" that corresponds to the ASCII code 67 that is in binary 01000011.

$$\mathbf{u} = \mathbf{rA} + \mathbf{e_1} = \begin{pmatrix} 0 & 1 & 1 & 1 \end{pmatrix} \cdot \begin{pmatrix} 85 & 53 & 10 & 53 \\ 15 & 37 & 24 & 31 \\ 81 & 51 & 3 & 61 \\ 12 & 26 & 29 & 15 \end{pmatrix} + \begin{pmatrix} 0 & 1 & -1 & -1 \end{pmatrix}$$
$$= \begin{pmatrix} 19 & 25 & 56 & 18 \end{pmatrix} + \begin{pmatrix} 0 & 1 & -1 & -1 \end{pmatrix}$$
$$= \begin{pmatrix} 19 & 26 & 55 & 17 \end{pmatrix}$$

$$\mathrm{v} = \mathbf{rt} + \mathbf{e_2} + \mu \left\lfloor \frac{q}{2} \right\rfloor = \begin{pmatrix} 0 & 1 & 1 & 1 \end{pmatrix} \cdot \begin{pmatrix} 5 \\ 39 \\ 85 \\ 40 \end{pmatrix} + \begin{pmatrix} 1 \end{pmatrix} + \begin{pmatrix} 44 \end{pmatrix}$$
$$= \begin{pmatrix} 75 \end{pmatrix} + \begin{pmatrix} 1 \end{pmatrix} + \begin{pmatrix} 44 \end{pmatrix}$$
$$= 120$$
$$= 31 \bmod q$$

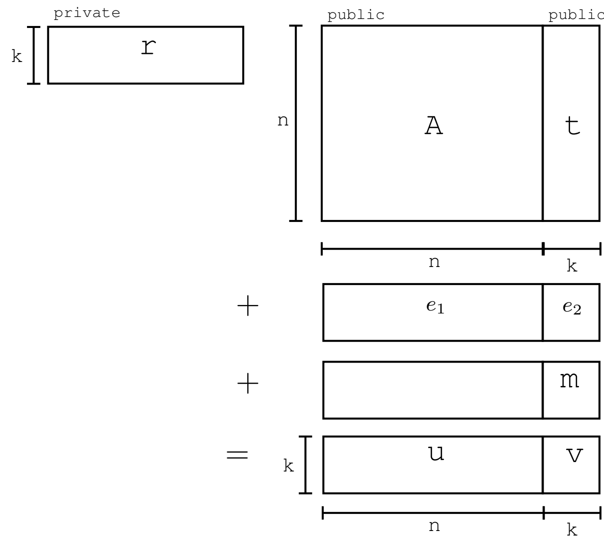


Figure 2.8: Encryption Function (computed by Bob)

Figure 2.9: Formula to compute v (computed by Bob)

Figure 2.10: Formula to compute u (computed by Bob)

2.5.3 Decryption

The goal of this method is to recover back the original message without any error. It is defined as:

$$\text{Dec}_{sk}(\mathbf{u}, \text{v}) = \begin{cases} 1 & \text{if v} - \mathbf{us} \mod q \in \left[\frac{q}{4}, \frac{3q}{4}\right] \\ 0 & \text{if v} - \mathbf{us} \mod q \ \textit{otherwise} \end{cases}$$

where

$$\text{v} - \mathbf{us} = \mathbf{rt} + \mathbf{e_2} + \mu \left\lfloor \frac{q}{2} \right\rfloor - \mathbf{us}$$

Reconciliation mechanism The error reconciliation is a mechanism used by two parties (i.e Alice and Bob) to concur on a secret bit, where they only share a common value up to an approximation factor. We will consider the Ding's [8] error reconciliation as a method to reconcile a message called m because was the first method proposed with basic rounding and has a good failure probability (that means obtaining a wrong output $m_{Alice} \neq m_{Bob}$) of 2^{-10}. There are other reconciliation mechanisms such as Peikert's error reconciliation [9] that has a smaller failure probability (2^{-16384}).

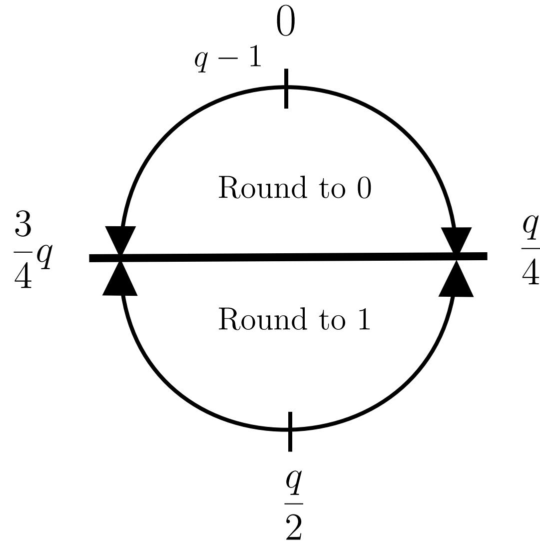


Figure 2.11: Rounding according to Ding's technique. Probability of failure is 2^{-10}

Example Considering the context of the previous example (the encryption computing by Bob), this time Alice uses $\mathbf{u}$ and v sent by Bob and her secret key $\mathbf{s}$ in order to decrypt the Bob's message:

$$
\begin{aligned}
\mathrm{v} - \mathbf{us} &= \mathbf{rt} + \mathbf{e_2} + \mu \left\lfloor \frac{q}{2} \right\rfloor - \mathbf{us} \\
&= 31 - \begin{pmatrix} 19 & 26 & 55 & 17 \end{pmatrix} \cdot \begin{pmatrix} 1 \\ 0 \\ 1 \\ 0 \end{pmatrix} \\
&= 31 - 74 \\
&= -43 \\
&= 46 \bmod q \\
&\xrightarrow[\frac{q}{4}<46<\frac{3}{4}q]{round} 1
\end{aligned}
$$

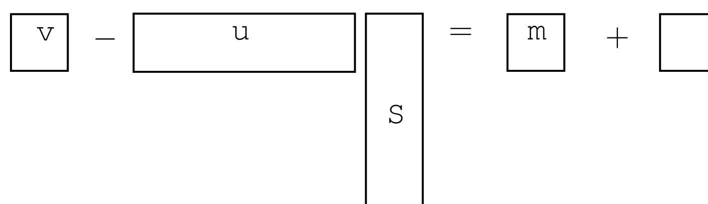


Figure 2.12: Decryption Function (computed by Alice)

2.5.4 NIST Standardization Competition

In order to evaluate and then standardize PQC schemes, on December 2016 the US National Institute of Standards and Technology (NIST) organized a formal call for PQC scheme proposals [10]. This agency in the same way standardized the algorithms SHA-1 in 1995 (already weakened in 2005) and SHA-2 in 2001.
On November 2017 (deadline for proposals) 69 submissions were accepted as complete and proper to participate. Most of the candidates were lattice-based and 2/3 were KEM/PKE and only 1/3 were signatures.
In six years of candidate analysis there were in total 3 rounds. The following tables show the three round lattice-based scheme candidates.
Table 2.1 shows the first round, table 2.2 corresponds to the second round and the third round is represented in the 2.3 table.

Winners of Round 3 After 6 years of competition, on July 5 2022, NIST announced the winners. For the lattice-based scheme type the winners are CRYSTALS-Kyber for the KEM/PKE. Instead, CRYSTALS-Dilithium and FALCON for the Signature system.

Table 2.1: In the first round the total lattice-based candidates account for 26 out of 64. Candidates "HILA5" and "Round2" were withdrawn and merged in "Round5" candidate.

PKE/KEM	Signature
- Compact LWE - **CRYSTALS-Kyber** - Ding Key Exchange - EMBLEM and R.EMBLEM - FrodoKEM - KCL - KINDI - LAC - LIMA - Lizard - LOTUS - NewHope - NTRUEncrypt - NTRU-HRSS-KEM - NTRU Prime - Odd Manhattan - Round5 - SABER - Three Bears - Titanium	- **CRYSTALS-Dilithium** - DRS - FALCON - pqNTRUSign - qTESLA

Table 2.2: Second round was announced on January 30, 2019. This time the lattice-based candidates accounted for 12 out of 26

PKE/KEM	Signature
- **CRYSTALS-Kyber** - FrodoKEM - LAC - NewHope - NTRU - NTRU Prime - Round5 - SABER - Three Bears	- **CRYSTALS-Dilithium** - FALCON - qTESLA

Table 2.3: Third round announced on July 22 2020, NIST included two new candidates (*) for the lattice-based type to be considered in the fourth round, they were "FrodoKEM" and "NTRU Prime". So now lattice-based accounted 7 out of 15

PKE/KEM	Signature
- **CRYSTALS-Kyber** - NTRU - SABER - FrodoKEM* - NTRU Prime*	- **CRYSTALS-Dilithium** - FALCON

Chapter 3

The Number Theoretic Transform (NTT) and its inverse (INTT)

3.1 NTT

The Number Theoretic Transform (NTT) is a special case of Discrete Fourier Transform (DFT) over a finite field [11, 12] that permit to perform more efficiently polynomial multiplication of high degree on integers sequences over a ring. This transform guarantees **full precision** because all computations are performed with integers. In contrast, classical FFT performed over the complex field might yield errors about rounding precision due to the use of floating points. NTT has a quasilinear complexity $O(nlogn)$ (reached using some tricks explained later in this chapter) where n is the length of polynomials so is faster than classic methods such as schoolbook algorithm that is the most simplest with quadratic complexity $O(n^2)$ or Karatsuba algorithm that uses the "divide-and-conquer" design paradigm with $O(n^{1.58})$ complexity or Toom-Cook algorithm that has a $O(n^r)$ cost where $r = log_k(2k-1)$.
For these particular features, NTT can be used for post-quantum cryptography.

3.1.1 Definitions

k-th primitive root of unity (ζ)

Let k be a positive integer, q a prime number so that $\mathbb{Z}_q$ is a finite field. Denote $\zeta \in \mathbb{Z}$, the k-th primitive root of unity is a number ζ which holds both conditions:

$$\zeta^k = 1 \quad \text{and} \quad \zeta^i \neq 1 \text{ where } i = 1,2,..,k-1$$

Monic Polynomial

A polynomial in a single variable (i.e x) that its leading coefficient (the coefficient of highest degree) is equal to 1. For example:

$$x^n + c_{n-1}x^{n-1} + \cdots + c_2x^2 + c_1x + c_0$$

with $n \geq 0$.

Irreducible Polynomial

Let $\phi(x)$ be a monic polynomial in the field $\mathbb{Z}$. It is called *irreducible* if it doesn't have nontrivial factors over $\mathbb{Z}[x]$.

Cyclotomic Polynomial

Let d be a positive integer, the d-th cyclotomic polynomial is an irreducicle polynomial defined over the field $\mathbb{Z}[x]$ that divides $x^d - 1$ but not $x^i - 1$ where $i < d$.

Representations of the polynomial rings $\mathbb{Z}_q[x]$

Let $\mathbb{Z}_q[x]$ be the polynomial ring over $\mathbb{Z}_q$ with quotient rings $\frac{\mathbb{Z}_q[x]}{<\phi(x)>}$. The $\phi(x)$ is a cyclotomic polynomial [13]. In particular, if $\deg \phi(x) = n$ is true, the element $\boldsymbol{a} \in \mathbb{Z}_q[x]/ < \phi(x) >$, can be represented both in the form of

$$\boldsymbol{a} = \sum_{i=0}^{n-1} a_i x^i$$

or

$$\boldsymbol{a} = (a_0, a_1, \ldots, a_{n-1})$$

where $a_i \in \mathbb{Z}_q$.

The Chinese Remainder Theorem (CRT)

Let $m_1, m_2, \ldots, m_r$ be integers > 1 with $r \geq 2$, that are pairwise coprime, which means $(m_i, m_j) = 1$ for $i \neq j$. Then if $a_1, \ldots, a_k$ are integers such that $0 \leq a_k < m_r$ for every r, the Chinese Remainder Theorem asserts that the following congruences

$$x \equiv a_1 \bmod m_1, \quad x \equiv a_2 \bmod m_2, \quad \ldots, \quad x \equiv a_r \bmod m_r,$$

has a unique solution $x \bmod N$, where $N = m_1 \cdot m_2 \cdots m_r$.

Example: The congruences

$$x \equiv 3 \bmod 11$$

$$x \equiv 21 \bmod 23$$

are satisfied when $x \equiv 113 \bmod 253$ and no other x. N is obtained as $253 = 11 \cdot 23$. To obtain each a_k value knowing x, just need to calculate:

$$113 \bmod 11 = 3$$

$$113 \bmod 23 = 21$$

Polynomial Multiplication and Convolution

If we consider two sequences with period q, their sequence numbers can be also interpreted as the coefficient of the polynomials $\boldsymbol{a}, \boldsymbol{b} \in \mathbb{Z}_q[X]/\phi(x)$, then computing the cyclic convolution ($\phi(x) = x^n - 1$) or computing the negacyclic convolution ($\phi(x) = x^n + 1$) of those sequences is the same as computing the polynomial multiplication $\boldsymbol{a} \cdot \boldsymbol{b}$ [14].

Therefore, depending on which reduction polynomial we use for the ring, there are mainly 2 types of convolution which are used in the context of lattice-based cryptography:

- **Cyclic (or Circular) convolution:** This convolution uses the $x^n - 1$ reduction polynomial so the cyclic convolution $\boldsymbol{c}$ is

$$\boldsymbol{c} = \boldsymbol{a} \cdot \boldsymbol{b} \in \mathbb{Z}_q[x]/\left(x^n - 1\right)$$

then

$$\boldsymbol{c} = \sum_{k=0}^{n-1} c_k x^k$$

where

$$c_k = \sum_{i=0}^{k} a_i b_{k-i} + \sum_{i=k+1}^{n-1} a_i b_{k+n-i} \bmod q \quad k = 0,1,\ldots,n-1$$

- **Negacyclic (or Negative wrapped) convolution:** Basically is very similar to cyclic convolution, but here the reduction polynomial is $x^n + 1$. So the negacyclic convolution $\boldsymbol{c}$ is defined as

$$\boldsymbol{c} = \boldsymbol{a} \cdot \boldsymbol{b} \in \mathbb{Z}_q[x]/\left(x^n + 1\right)$$

then

$$\boldsymbol{c} = \sum_{k=0}^{n-1} c_k x^k$$

where

$$c_k = \sum_{i=0}^{k} a_i b_{k-i} - \sum_{i=k+1}^{n-1} a_i b_{k+n-i} \bmod q \quad k = 0{,}1, \ldots, n-1$$

The $\mathbb{Z}_q[x]/\left(x^n - 1\right)$ and $\mathbb{Z}_q[x]/\left(x^n + 1\right)$ are mainly used in lattice-based schemes. For the purpose of studying the Kyber scheme we will only focus to describe the last one (Negative wrapped convolution).

3.1.2 NTT-based Polynomial Multiplication

Remembering that NTT is a transformation and not an algorithm, there are many variants in which the NTT can be implemented. Only the Negacyclic Convolution-based NTT will be described because is relevant for Kyber scheme. The most important concept is that NTT can be used to compute negacyclic (or linear, or cyclic) convolutions [15] that corresponds to compute a polynomial multiplication.

Negacyclic convolution-based NTT

Let q a prime number that satisfies $q \equiv 1 \bmod 2n$ such that exist a 2n-th root of unity ζ_{2n} in $\mathbb{Z}_q$. The zeta vector is defined as follow:

$$\boldsymbol{\zeta} = \left(1, \zeta_{2n}, \zeta_{2n}^2, \ldots, \zeta_{2n}^{n-1}\right)$$

$$\boldsymbol{\zeta}^{-1} = \left(1, \zeta_{2n}^{-1}, \zeta_{2n}^{-2}, \ldots, \zeta_{2n}^{-(n-1)}\right)$$

The pointwise multiplication (that means the product of each element of vector $\boldsymbol{a}$ by the corresponding element of the zeta vector $\boldsymbol{\zeta}$) is denoted with the symbol $\circ$, so

$$\overline{\boldsymbol{a}} = \boldsymbol{\zeta} \circ \boldsymbol{a} \quad \text{where } \bar{a}_i = \zeta_{2n}^i a_i$$

$$\boldsymbol{a} = \boldsymbol{\zeta}^{-1} \circ \overline{\boldsymbol{a}} \quad \text{where } a_i = \zeta_{2n}^{-i} \bar{a}_i$$

Then the trasformation is denoted as follow

$$\hat{\boldsymbol{a}} = \mathrm{NTT}^{\zeta}(\boldsymbol{a}) = \mathrm{NTT}(\boldsymbol{\zeta} \circ \boldsymbol{a})$$

$$\boldsymbol{a} = \mathrm{INTT}^{\zeta^{-1}}(\hat{\boldsymbol{a}}) = \boldsymbol{\zeta}^{\mathbf{-1}} \circ \mathrm{INTT}(\hat{\boldsymbol{a}})$$

considering $\omega_n = \zeta_{2n}^2$, the single j-th element in $\hat{\boldsymbol{a}}$ can be expanded as

$$\hat{a}_j = \sum_{i=0}^{n-1} a_i \zeta_{2n}^i \omega_n^{ij} \bmod q \quad j = 0,1,\ldots,n-1$$

and the inverse can be written as:

$$a_i = n^{-1} \zeta_{2n}^{-i} \sum_{j=0}^{n-1} \hat{a}_j \omega_n^{-ij} \bmod q \quad i = 0,1,\ldots,n-1.$$

At this point, there are two important properties:

Property 1:

$$\boldsymbol{a} = \mathrm{INTT}^{\zeta^{-1}} \left(\mathrm{NTT}^{\zeta}(\boldsymbol{a}) \right)$$

Property 2:

$$\mathrm{NTT}^{\zeta}(\boldsymbol{c}) = \mathrm{NTT}^{\zeta}(\boldsymbol{a}) \circ \mathrm{NTT}^{\zeta}(\boldsymbol{b})$$

Therefore, in order to compute the negacyclic convolution $\boldsymbol{c} = \boldsymbol{a} \cdot \boldsymbol{b} \in \mathbb{Z}_q[x]/\left(x^n + 1\right)$ we can use the combination of the previous two properties so that $\boldsymbol{c}$ is obtained as follow:

$$\boldsymbol{c} = \mathrm{INTT}^{\zeta^{-1}} \left(\mathrm{NTT}^{\zeta}(\boldsymbol{a}) \circ \mathrm{NTT}^{\zeta}(\boldsymbol{b}) \right) \tag{3.1}$$

Computing formula 3.1 without any *trick* has a $O\left(n^2\right)$ complexity, so until now no improvements in terms of performance are obtained compared to using classic algorithms to compute polynomial multiplications.

3.1.3 FFT Trick for Negacyclic-based NTT $\in \mathbb{Z}_q(x)/(x^n + 1)$

The Fast Fourier transform (FFT) is an algorithm to compute in a fast way the discrete fourier transform (or its inverse) of a sequence. As mention in the previous paragraph, NTT needs a "trick" in order to perform better than $O\left(n^2\right)$.
In this section I will describe the case of FFT Trick suited for Negacyclic-based NTT $\in \mathbb{Z}_q(x)/(x^n + 1)$ because is relevant for Kyber scheme.

The Trick

If n is a power of 2, q is a prime number and $q \cong 1$ (mod 2n), exists an efficient way to easily compute NTT^{ζ} and $INTT^{\zeta}$ according to the work [16]. Considering the Chinese Remainder Theorem in ring form [17], for polynomial rings $\mathbb{Z}_q[x]/(x^n + 1)$ exist the following isomorphism:

$$\Phi : \mathbb{Z}_q[x]/\left(x^n + 1\right) \cong \mathbb{Z}_q[x]/\left(x^{\frac{n}{2}} - \zeta_{2n}^{\frac{n}{2}}\right) \times \mathbb{Z}_q[x]/\left(x^{\frac{n}{2}} + \zeta_{2n}^{\frac{n}{2}}\right) \tag{3.2}$$

because $\zeta_{2n}^{n} = -1 \bmod q$ so the following equality holds and can be used in 3.2:

$$(x^n + 1) = (x^n - \zeta_{2n}^{n}) = (x^{\frac{n}{2}} - \zeta_{2n}^{\frac{n}{2}})(x^{\frac{n}{2}} + \zeta_{2n}^{\frac{n}{2}})$$

If we keep repeating recursively down this polynomial decomposition until reach a degree-0 polynomial, we can express the CRT isomorphism 3.2 as:

$$\mathbb{Z}_q[x]/\left(x^n + 1\right) \cong \prod_{i=0}^{n-1} \mathbb{Z}_q[x]/\left(x - \zeta_{2n}^{2\,\mathrm{brv}_n(i)+1}\right).$$

where the operator $\mathrm{brv}_n(i)$ is the bit reversal of the unsigned n-bit integer i, and this ordering is used in order to be compatible with AVX instructions [18]. The Figure 3.1 represents a general binary tree of such polynomial decomposition that contains $0 \leq k < log(n)$ levels.

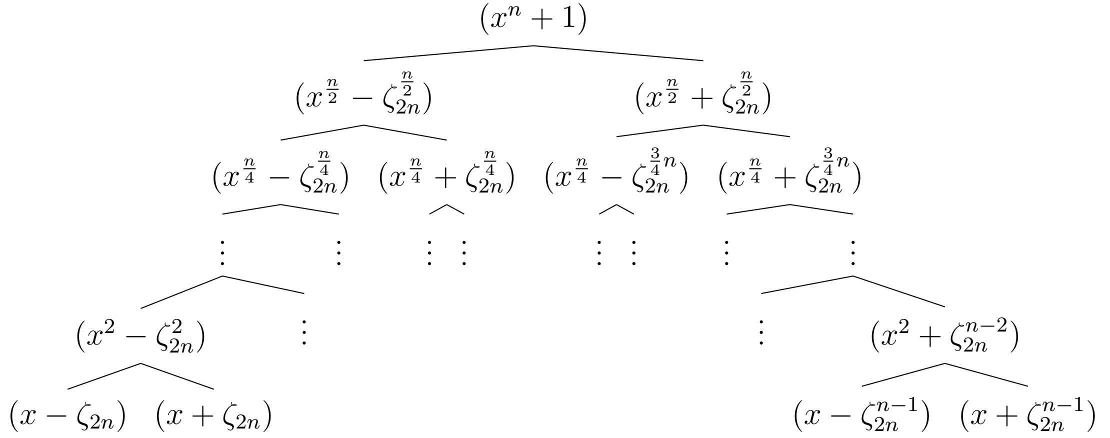


Figure 3.1: Binary Tree representing the Chinese Remainder Theorem map of FFT Trick over $\mathbb{Z}_q[x]/(x^n + 1)$

Let $\mathbf{a} \in \mathbb{Z}_q[x]/\left(x^n + 1\right)$, the **forward transform** $\Phi\left(\mathbf{a}\right)$ for each binary decomposition is computed as following:

$$\Phi\left(\sum_{i=0}^{n-1} a_i x^i\right) = (\boldsymbol{a}_L, \boldsymbol{a}_R) = \left(\sum_{i=0}^{\frac{n}{2}-1}\left(a_i + \zeta^{\frac{n}{2}} \cdot a_{i+\frac{n}{2}}\right)x^i, \sum_{i=0}^{\frac{n}{2}-1}\left(a_i - \zeta^{\frac{n}{2}} \cdot a_{i+\frac{n}{2}}\right)x^i\right)$$

In the previous equation, the addend $(a_i + \zeta^{\frac{n}{2}} \cdot a_{i+\frac{n}{2}})$ of the summation from term $\mathbf{a}_L$ is obtained thanks to the fact that we can collect as a common factor the terms having the same degree because

$$X^{\frac{n}{2}} = \zeta^{\frac{n}{2}}$$

such that

$$a_i X^i = a_i \zeta^{\frac{n}{2}} X^{i-\frac{n}{2}}$$

then we can collect as a common factor the terms having the same degree. For example, for $i = 130$ the terms $a_2 X^2$ and $a_{130}\zeta^{128} X^{130-128}$ can be collected by X^2. For instance, in the first layer of the NTT algorithm $\frac{n}{2} = 128$, so the corresponding decomposition is:

$$\mathbf{a}_L^{(1)} = \left(a_0 + \zeta^{128} a_{128}\right) + \left(a_1 + \zeta^{128} a_{129}\right) X + \left(a_2 + \zeta^{128} a_{130}\right) X^2 + \dots$$

The same is for the $\mathbf{a}_R$ because

$$X^{\frac{n}{2}} = -\zeta^{\frac{n}{2}}$$

so to reduce modulo this polynomial we have to multiply with $-\zeta^{\frac{n}{2}}$. So, again if $\frac{n}{2} = 128$ the corresponding decomposition is:

$$\mathbf{a}_R^{(1)} = \left(a_0 - \zeta^{128} a_{128}\right) + \left(a_1 - \zeta^{128} a_{129}\right) X + \left(a_2 - \zeta^{128} a_{130}\right) X^2 + \dots$$

Since such multiplication between a_i and $\zeta^{\frac{n}{2}}$ appears in both $\mathbf{a}_L$ and $\mathbf{a}_R$, we can multiply only once and therefore saving machine cycles for that computation. The Cooley-Tukey (CT) butterfly [19] is an efficient method which takes advantage of this feature, figure 3.2 shows that process.

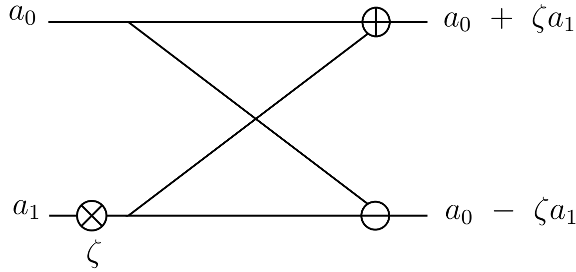


Figure 3.2: Cooley-Tukey (CT) butterfly

The **inverse transform** $\Phi^{-1}(\mathbf{a}_L, \mathbf{a}_R)$ for each binary decomposition is defined as:

$$\Phi^{-1}\left(\sum_{i=0}^{\frac{n}{2}-1} a_i' x^i, \sum_{i=0}^{\frac{n}{2}-1} a_i'' x^i\right) = \sum_{i=0}^{\frac{n}{2}-1} \frac{1}{2}\left(a_i' + a_i''\right) x^i + \sum_{i=0}^{\frac{n}{2}-1} \frac{1}{2}\zeta^{-\frac{n}{2}}\left(a_i' - a_i''\right) x^{i+\frac{n}{2}}$$

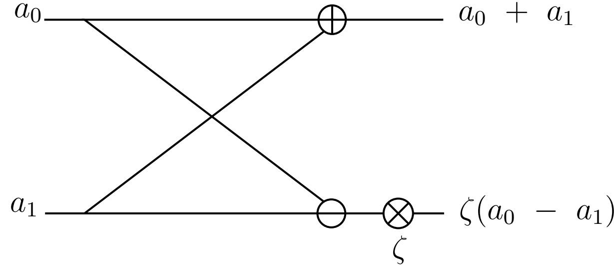


Figure 3.3: Gentleman-Sande (GS) butterfly

In order to explain the formula, considering again the first layer in which $\frac{n}{2} = 128$ and the first iteration ($i = 0$):

$$a^{'} = a_{L,0}^{(1)} = a_0 + \zeta^{128} a_{128}$$
$$a^{''} = a_{R,0}^{(1)} = a_0 - \zeta^{128} a_{128}$$

To find $\mathbf{a}$ that is the inverse of $(\mathbf{a}_L, \mathbf{a}_R)$, we can consider these observations and then obtain the final formulas:

$$a_{L,0}^{(1)} + a_{R,0}^{(1)} = a_0 + \cancel{\zeta^{128} a_{128}} + a_0 - \cancel{\zeta^{128} a_{128}}$$
$$2a_0 = a_{L,0}^{(1)} + a_{R,0}^{(1)}$$
$$a_0 = \frac{1}{2}\left(a_{L,0}^{(1)} + a_{R,0}^{(1)}\right)$$
$$a_{L,0}^{(1)} - a_{R,0}^{(1)} = \cancel{a_0} + \zeta^{128} a_{128} - \cancel{a_0} + \zeta^{128} a_{128}$$
$$2\zeta^{128} a_{128} = a_{L,0}^{(1)} - a_{R,0}^{(1)}$$
$$a_{128} = \frac{1}{2}\zeta^{-128}\left(a_{L,0}^{(1)} - a_{R,0}^{(1)}\right)$$

Since division by 2 that appears for every iteration, it can be accumulated as 2^{-k} where the k represents the number of layers, so it can be easily computed at the end (or at the beginning) for skipping useless divisions.

This way of computing the inverse transform is known as Gentlemen-Sande butterfly (GS) [20], the figure 3.3 depicts these operations.
Both algorithms (CT and GS) are called "butterfly" because if we used segments to represent the phases in which addition, subtraction and multiplication are performed, these draw the wings of a butterfly.

Complexity At this point, the transformation of the polynomials in the NTT domain (using the FFT Trick) requires the recursion along the binary tree (generated by the application of the CRT) which has a number of levels equal to $log_2(n)$

of which each perform n multiplications with the corresponding zeta coefficient for that level. Once the transformation is finished, that is, when the last level of this tree is reached, the operations can be performed in a "pointwise" way that is component by component of the two transformed polynomials and finally compute the inverse transformation (INTT) to get the complete polynomial multiplication. Consequently, the recursion of the binary tree to obtain the complete NTT transformation of the **a** and **b** vectors implies a complexity equal to $O(nlog(n))$, then the *pointwise* operations implies a $O(n)$ complexity and finally the inverse transformation (INTT) requires again $O(nlog(n))$, therefore, considering the worst complexity, the final complexity of computing $\boldsymbol{c} = \boldsymbol{a} \cdot \boldsymbol{b} \in \mathbb{Z}_q[x]/\left(x^n + 1\right)$ with NTT considering this trick is equal to $O(nlog(n))$.

Chapter 4

Kyber

In this chapter we will describe the specifications and mode of operation of CRYSTALS-KYBER.

4.1 Preliminaries

Before discussing Kyber, it's important to introduce some terminology that we use in this chapter, and methods for evaluating the level of security of cryptographic systems as well.

4.1.1 Oracle

In general, an oracle can be conceptualized as a black box capable of solving specific problems. For our purposes, an oracle is an abstract entity that provides encryption/decryption capabilities to an adversary. For instance, an encryption oracle can accept plaintexts as input and return the corresponding ciphertexts. A decryption oracle, instead, accepts as input parameter ciphertexts, and returns the corresponding plaintexts

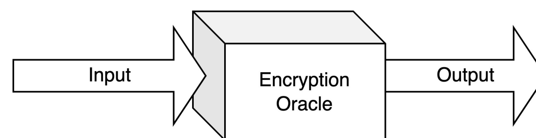


Figure 4.1: Example of an Encryption Oracle

4.1.2 Security Goals

Indistinguishability (IND) is a cryptographic property where an attacker cannot distinguish between two different encrypted data even if he has access to certain information or perform multiple operations on data. Hence, indistinguishability plays a crucial role in preserving the confidentiality of encrypted communications.

Nowadays, we aim for the highest level of security assurance, therefore it's very important to classify the security level of a scheme in terms of indistinguishability.

We can establish the level of security by assessing how an adversary can win a game against a challenger. For the sake of simplicity, a schema is considered secure if the adversary's advantage is negligible. An adversary has a negligible advantage if he cannot perform significantly better than random guessing. Among all types of security goals, only two will be described because they are used within the Kyber scheme.

4.1.3 IND-CPA

A security goal which ensures Indistinguishability under Chosen Plaintext Attack (IND-CPA) guarantees the confidentiality of encrypted data even when an adversary has access to an encryption oracle. This goal can be described with the following game:

1. The challenger creates a key pair (PK, SK) using a security parameter k (e.g., a key size in bits) and shares only the PK with the adversary.

2. The adversary could carry out a limited number of encryption operations or other actions within a polynomial time frame

3. The adversary chooses two plaintexts (m_0,m_1) and sends them to the challenger.

4. The challenger randomly chooses a bit, $b \xleftarrow{\$} \{0,1\}$, and then transmits the challenge ciphertext $c^* \leftarrow \text{Enc}\,(k, m_b^*)$ to the adversary.

5. The adversary can freely conduct as many extra computations or encryptions as they desire. Finally, he outputs b', a guess for the value of b. If $b = b'$, the adversary wins.

Considering this game, a cryptosystem is considered IND-CPA secure if any adversary, operating within probabilistic polynomial time, possesses only a negligible advantage over random guessing. An adversary is said to have a negligible advantage if it wins the above game with probability

$$\frac{1}{2} + \epsilon(k)$$

where $\epsilon()$ is a negligible function and k the security parameter. Although the original definition is tailored for asymmetric key cryptosystems, it can be modified for symmetric key scenarios by substituting the public key encryption function with an encryption oracle.

4.1.4 IND-CCA

In this scenario, the adversary has access to both encryption and decryption oracle. They can use the encryption oracle to encrypt messages with the public key and the decryption oracle to decrypt any given ciphertext, revealing the original plaintext. This goal can be described with the following game:

1. The challenger creates a key pair (PK, SK) using a security parameter k (e.g., a key size in bits) and shares only the PK with the adversary.

2. The adversary can execute any number of calls to the encryption and decryption oracle using arbitrary ciphertexts, as well as conduct other operations as desired

3. The adversary chooses two plaintexts (m_0,m_1) and sends them to the challenger.

4. The challenger randomly chooses a bit $b \xleftarrow{\$} \{0,1\}$ and then transmits the challenge ciphertext $c^* \leftarrow \mathrm{Enc}\,(k, m_b^*)$ to the adversary.

5. The adversary can freely conduct as many extra computations or encryptions as they desire. At this point, there are two variants to take in consideration:

   (a) The non-adaptive case (IND-CCA1): the adversary cannot perform additional calls to the decryption oracle.

   (b) The adaptive case (IND-CCA2): is allowed to continue making calls to the decryption oracle, but may not submit the challenge ciphertext c^*.

   Finally, he outputs b', a guess for the value of b. If $b = b'$, the adversary wins.

A schema is considered IND-CPA secure if any adversary, operating within probabilistic polynomial time, possesses only a negligible advantage over random guessing.

4.1.5 Implications

IND-CCA2 is the more stringent security criterion. For instance, if we consider a IND-CCA2 secure scheme, it also guarantees IND-CCA1 and IND-CPA security.

IND-CCA2 $\implies$ IND-CCA1 $\implies$ IND-CPA

4.2 CRYSTALS

CRYSTALS [18] is the acronym for Cryptographic Suite for Algebraic Lattices. This suite was supported by the European Commission programs, the Swiss National Science Foundation, the Netherlands Organization for Scientific Research and the German Research Foundation (DFG). This suite contains two cryptographic schemes which rely on hard problems over module lattices:

- Kyber: a IND-CCA2 secure Key Encapsulation Mechanism (KEM)
- Dilithium: a digital signature algorithm

For the purposes of this work we will only study and describe CRYSTALS-KYBER which uses the key encapsulation mechanism as a cryptographic technique in order to establish a secure communication. This mechanism is similar to the Public Key Encryption (PKE) because requires the generation of public and private keys, but KEM is mainly intended for distributing secret keys between parties. More in detail, KEM mechanism encapsulates a randomly generated secret key within a ciphertext, which is then sent to the receiver, so he can then decrypt the ciphertext to obtain the secret key for then use in symmetric-key encryption.

4.2.1 Functions and Parameters Employed

Kyber offers the option to adjust different security levels by selecting one of 3 parameters: Kyber-512, Kyber-768 and Kyber-1024 listed in table 4.1. These 3 names come from the multiplication between n and q (e.g., Kyber-768 = $256 \cdot 3$). Varying these levels of security, users can select the most a suitable configuration based on their specific requirements. The parameter k, instead, indicates the size of the lattice.

The notation B^k used in the next Kyber's algorithms, represents the set of byte arrays of length k where each element is an integer of 8-bit so in the set of $0, ...,255$.

The expression $(a||b)$ denotes the concatenation of two arrays, a and b.

In table 4.2 are listed all the cryptographic functions that Kyber uses in its algorithms. In particular, H and G are hash functions, XOF is the Extendable Output Function which generates a random array with an arbitrary length, and KDF that means Key Derivation Function which generates secret keys from a master key and an input string.

There is also a variant of Kyber called "90s" which uses instead symmetric primitives (AES and SHA-256) that are standardized by NIST and accelerated in hardware across a broad spectrum of platforms (e.g., recent Intel, AMD, and ARM processors).

Table 4.1: Kyber Parameters

	n	k	q	η_1	η_2	d_u	d_v	δ
KYBER-512	256	2	3329	3	2	10	4	2^{-139}
KYBER-768	256	3	3329	2	2	10	4	2^{-164}
KYBER-1024	256	4	3329	2	2	11	5	2^{-174}

Table 4.2: Kyber Functions

	Modern	'90 variant
XOF	SHAKE-128	AES-256 in CTR mode
H	SHA3-256	SHA-256
G	SHA3-512	SHA-512
PRF(s,b)	SHAKE-256(s||b)	AES-256 in CTR mode
KDF	SHAKE-256	SHAKE-256

4.2.2 Kyber's Polynomials and Coefficients

In Kyber, rings $\mathbb{Z}[X]/(X^n + 1)$ and $\mathbb{Z}_q[X]/(X^n + 1)$ are respectively denoted by R and R_q and considering $n = 256$, n= 9, and $q = 3329$ such that $X^n + 1$ is the 2^9th cyclotomic polynomial.

Kyber mainly uses the range $\left\{-\left\lceil\frac{q}{2}\right\rceil + 1, \dots, \left\lfloor\frac{q}{2}\right\rfloor\right\}$ which is congruent to 0, ..., $q - 1$ as a representative set of $\mathbb{Z}_{\shortparallel}$

Elements in R or R_q are expressed in regular font letters. Vectors with coefficients in R or R_q are denoted as letters in bold lower-case. Bold upper-case letters are used to indicate matrices. For instance, $\mathbf{v}$ is a vector and $\mathbf{A}$ is a matrix and the notation $\mathbf{v}^T$ (or $\mathbf{A}^T$) represents their transpose.

4.3 Security Approach

Kyber is an IND-CCA2-secure key-encapsulation mechanism (KEM). The security of Kyber is based on the hardness of solving the learning-with-errors problem in module lattices and this is called Module-Learning With Errors (MLWE). Kyber's security approach employs a dual-phase methodology.

- Kyber.CPAPKE
- Kyber.CCAKEM

This approach guarantees efficiency and the security against both classical and quantum attacks.

4.4 Kyber.CPAPKE

Overall, this technique consists on encrypting fixed length messages of 32-bytes using the INDCPA-secure public-key encryption scheme.

This phase cannot be used as a standalone encryption solution as it does not guarantee all the necessary standard security requirements. The output of this function is then used as input for the Kyber.CCAKEM.

4.4.1 Key Generation

The Key Generation algorithm in figure 4.2 has no input arguments, but it's internally parameterized by the parameters n, k, q, η_1, η_2, d_u, d_v. It returns the secret and public key. The length of each key depends on the chosen Kyber version as indicated in the table 4.3.

Table 4.3: Secret and Public key size depend on chosen Kyber's version

	Secret-key Bytes	Public-key Bytes
KYBER-512	$12 \cdot k \cdot n/8 = 768$	$12 \cdot k \cdot n/8 + 32 = 800$
KYBER-768	$12 \cdot k \cdot n/8 = 1152$	$12 \cdot k \cdot n/8 + 32 = 1184$
KYBER-1024	$12 \cdot k \cdot n/8 = 1536$	$12 \cdot k \cdot n/8 + 32 = 1568$

From line 4 to 8 of the depicted pseudocode there is a double loop that populate the $\mathbf{A}$ matrix of polynomials of dimension $k \times k$. Each polynomial is represented by a vector of 256 coefficients, because the degree of the polynomial is 256, and they are defined in $\mathbb{Z}_{||}$, then each coefficient can fit in a 16-byte integer data type. On line 6, the XOF function uses Keccak mechanism that generates a pseudo-random byte-stream of arbitrary length for each iteration. Then, to obtain an output in the R_q^n domain, it is necessary to apply the Kyber's Parse 4.3 function where each iteration produces a single coefficient of the array of size 256.

The Centered Binomial Distribution function 4.4 and PRF are used for the random generation of the secret $\mathbf{s}$ and the error $\mathbf{e}$ vector of k polynomials (lines 10 and 14).

Vectors $\mathbf{s}$ and $\mathbf{e}$ are transformed in the NTT domain (lines 17 and 18) in order to have a fast pointwise multiplication (line 19) between matrix $\mathbf{A}$ and $\mathbf{s}$. On lines 20 and 21, starting from the most inner operation, $\hat{\boldsymbol{t}}$ and $\hat{\boldsymbol{s}}$ are reduced in order to be in $\left\{-\frac{(q-1)}{2}, ..., \frac{q-1}{2}\right\}$ congruent to a mod q using the Barret's Reduction function. Only for $\hat{\boldsymbol{t}}$, the previous output is then concatenated with the public-seed which was previously used in order to generate matrix $\mathbf{A}$.

Finally, both partial outcomes are encoded to an array of 384 bytes ($32 \cdot l$ where $l = 12$).

Algorithm 4 KYBER.CPAPKE.KeyGen(): key generation

Output: Secret key $sk \in \mathcal{B}^{12 \cdot k \cdot n/8}$
Output: Public key $pk \in \mathcal{B}^{12 \cdot k \cdot n/8+32}$
1: $d \leftarrow \mathcal{B}^{32}$
2: $(\rho, \sigma) := \mathrm{G}(d)$
3: $N := 0$
4: **for** i from 0 to $k-1$ **do** ▷ Generate matrix $\hat{\mathbf{A}} \in R_q^{k \times k}$ in NTT domain
5: **for** j from 0 to $k-1$ **do**
6: $\hat{\mathbf{A}}[i][j] := \mathsf{Parse}(\mathsf{XOF}(\rho, j, i))$
7: **end for**
8: **end for**
9: **for** i from 0 to $k-1$ **do** ▷ Sample $\mathbf{s} \in R_q^k$ from B_{η_1}
10: $\mathbf{s}[i] := \mathsf{CBD}_{\eta_1}(\mathsf{PRF}(\sigma, N))$
11: $N := N + 1$
12: **end for**
13: **for** i from 0 to $k-1$ **do** ▷ Sample $\mathbf{e} \in R_q^k$ from B_{η_1}
14: $\mathbf{e}[i] := \mathsf{CBD}_{\eta_1}(\mathsf{PRF}(\sigma, N))$
15: $N := N + 1$
16: **end for**
17: $\hat{\mathbf{s}} := \mathsf{NTT}(\mathbf{s})$
18: $\hat{\mathbf{e}} := \mathsf{NTT}(\mathbf{e})$
19: $\hat{\mathbf{t}} := \hat{\mathbf{A}} \circ \hat{\mathbf{s}} + \hat{\mathbf{e}}$
20: $pk := (\mathsf{Encode}_{12}(\hat{\mathbf{t}} \bmod^+ q) \| \rho)$ ▷ $pk := \mathbf{As} + \mathbf{e}$
21: $sk := \mathsf{Encode}_{12}(\hat{\mathbf{s}} \bmod^+ q)$ ▷ $sk := \mathbf{s}$
22: **return** (pk, sk)

Figure 4.2: Kyber.CPAPKE.KeyGen() [18]

Algorithm 1 Parse: $\mathcal{B}^* \rightarrow R_q^n$

Input: Byte stream $B = b_0, b_1, b_2 \cdots \in \mathcal{B}^*$
Output: NTT-representation $\hat{a} \in R_q$ of $a \in R_q$
$i := 0$
$j := 0$
while $j < n$ **do**
$d_1 := b_i + 256 \cdot (b_{i+1} \bmod^+ 16)$
$d_2 := \lfloor b_{i+1}/16 \rfloor + 16 \cdot b_{i+2}$
if $d_1 < q$ **then**
$\hat{a}_j := d_1$
$j := j + 1$
end if
if $d_2 < q$ **and** $j < n$ **then**
$\hat{a}_j := d_2$
$j := j + 1$
end if
$i := i + 3$
end while
return $\hat{a}_0 + \hat{a}_1 X + \cdots + \hat{a}_{n-1} X^{n-1}$

Figure 4.3: Pseudocode of the Parse [18] function

Encryption

Encryption algorithm in figure 4.5 requires three inputs:

Algorithm 2 $\mathsf{CBD}_\eta \colon \mathcal{B}^{64\eta} \to R_q$

Input: Byte array $B = (b_0, b_1, \ldots, b_{64\eta-1}) \in \mathcal{B}^{64\eta}$
Output: Polynomial $f \in R_q$
$(\beta_0, \ldots, \beta_{512\eta-1}) \coloneqq \mathsf{BytesToBits}(B)$
for i from 0 to 255 **do**
  $a \coloneqq \sum_{j=0}^{\eta-1} \beta_{2i\eta+j}$
  $b \coloneqq \sum_{j=0}^{\eta-1} \beta_{2i\eta+\eta+j}$
  $f_i \coloneqq a - b$
end for
return $f_0 + f_1X + f_2X^2 + \cdots + f_{255}X^{255}$

Figure 4.4: Pseudocode of the centered binomial distribution (CBD) [18]

- pk the public key sent from the user which executed the KeyGen()
- m the messagge
- r random coins

The structure of this function is quite similar to the previous one, but in this case it retrieves the parameters $\hat{\boldsymbol{t}}$ (line 2) and $\hat{\mathbf{A}}^T$ (line 6) transmitted within the public key. Hence, a decode function is implied to obtain $\hat{\boldsymbol{t}}$ from pk (line 2). Subsequently, $\mathbf{r}$ (line 10), $\boldsymbol{e_1}$ (line 14), and e_2 are created in the same manner as $\mathbf{A}$ and $\mathbf{s}$ were created in the KeyGen() function, but in this case the seed, required to generate them, corresponds to the random coin passed as a parameter to this function. Vector $\mathbf{r}$ is transformed in the NTT domain (line 18). At this point, two inverse NTT transformation are performed to extract $\mathbf{u}$ (line 19) and v (line 20). More in detail, since message m is 256 bits (32 bytes), the Decode function on line 20 deserializes it into a polynomial. After this decoding phase, the algorithm executes the Decompress function to create error tolerance gaps in the M-LWE scheme:

$$\text{Decompress}_q(x, d) = \left\lceil \left(\frac{q}{2^d} \cdot x \right) \right\rfloor \tag{4.1}$$

Finally, $c1$ and $c2$ (lines 21 and 22) are obtained by applying the Compress function which balances the effect of the previous Decompression function:

$$\text{Compress}_q(x, d) = \left\lceil \left(\frac{2^d}{q} \cdot x \right) \right\rfloor \bmod {}^+ 2^d \tag{4.2}$$

The Encode() function is used to serialize the polynomials to arrays of $(32 \cdot d_u)$ or $(32 \cdot d_v)$ bytes. Finally, the output ciphertext corresponds to the concatenation of $c1$ and $c2$.

Algorithm 5 KYBER.CPAPKE.Enc(pk, m, r): encryption

Input: Public key $pk \in \mathcal{B}^{12 \cdot k \cdot n/8+32}$
Input: Message $m \in \mathcal{B}^{32}$
Input: Random coins $r \in \mathcal{B}^{32}$
Output: Ciphertext $c \in \mathcal{B}^{d_u \cdot k \cdot n/8 + d_v \cdot n/8}$

1: $N := 0$
2: $\hat{\mathbf{t}} := \mathsf{Decode}_{12}(pk)$
3: $\rho := pk + 12 \cdot k \cdot n/8$
4: **for** i from 0 to $k-1$ **do** ▷ Generate matrix $\hat{\mathbf{A}} \in R_q^{k \times k}$ in NTT domain
5:     **for** j from 0 to $k-1$ **do**
6:         $\hat{\mathbf{A}}^T[i][j] := \mathsf{Parse}(\mathsf{XOF}(\rho, i, j))$
7:     **end for**
8: **end for**
9: **for** i from 0 to $k-1$ **do** ▷ Sample $\mathbf{r} \in R_q^k$ from B_{η_1}
10:     $\mathbf{r}[i] := \mathsf{CBD}_{\eta_1}(\mathsf{PRF}(r, N))$
11:     $N := N + 1$
12: **end for**
13: **for** i from 0 to $k-1$ **do** ▷ Sample $\mathbf{e}_1 \in R_q^k$ from B_{η_2}
14:     $\mathbf{e}_1[i] := \mathsf{CBD}_{\eta_2}(\mathsf{PRF}(r, N))$
15:     $N := N + 1$
16: **end for**
17: $e_2 := \mathsf{CBD}_{\eta_2}(\mathsf{PRF}(r, N))$ ▷ Sample $e_2 \in R_q$ from B_{η_2}
18: $\hat{\mathbf{r}} := \mathsf{NTT}(\mathbf{r})$
19: $\mathbf{u} := \mathsf{NTT}^{-1}(\hat{\mathbf{A}}^T \circ \hat{\mathbf{r}}) + \mathbf{e}_1$ ▷ $\mathbf{u} := \mathbf{A}^T\mathbf{r} + \mathbf{e}_1$
20: $v := \mathsf{NTT}^{-1}(\hat{\mathbf{t}}^T \circ \hat{\mathbf{r}}) + e_2 + \mathsf{Decompress}_q(\mathsf{Decode}_1(m), 1)$ ▷ $v := \mathbf{t}^T\mathbf{r} + e_2 + \mathsf{Decompress}_q(m, 1)$
21: $c_1 := \mathsf{Encode}_{d_u}(\mathsf{Compress}_q(\mathbf{u}, d_u))$
22: $c_2 := \mathsf{Encode}_{d_v}(\mathsf{Compress}_q(v, d_v))$
23: **return** $c = (c_1 \| c_2)$ ▷ $c := (\mathsf{Compress}_q(\mathbf{u}, d_u), \mathsf{Compress}_q(v, d_v))$

Figure 4.5: Kyber.CPAPKE.Enc(pk, m, r) [18]

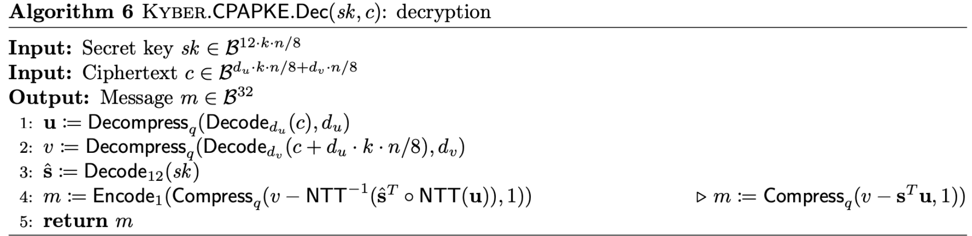
4.4.2 Decryption

The Decryption function in figure 4.6 takes as arguments the secret key and the ciphertext, and it returns the decrypted message. Vectors $\mathbf{u}$ and v (lines 1 and 2) are recovered from the ciphertext and $\hat{\mathbf{s}}$ (line 3) from the secret key. The recovering process involves an initial decoding operation followed by a decompression process (line 4). At this point, let denote d variable as the Compress argument, this variable is obtained applying the following formula:

$$\begin{aligned} d &= v - s \cdot \boldsymbol{u} \\ d &= v - NTT^{-1}(\hat{\boldsymbol{s}^T} \circ NTT(\boldsymbol{u})) \end{aligned} \tag{4.3}$$

Therefore, Compress(d,1) function is used in line 4 to decrypt each coefficient of the result of the previous equation to:

- 0: if $|x| < \frac{q}{4}$
- 1: if $|x| > \frac{q}{4}$



Algorithm 6 KYBER.CPAPKE.Dec(sk, c): decryption

Input: Secret key $sk \in \mathcal{B}^{12 \cdot k \cdot n/8}$
Input: Ciphertext $c \in \mathcal{B}^{d_u \cdot k \cdot n/8 + d_v \cdot n/8}$
Output: Message $m \in \mathcal{B}^{32}$
1: $\mathbf{u} := \mathsf{Decompress}_q(\mathsf{Decode}_{d_u}(c), d_u)$
2: $v := \mathsf{Decompress}_q(\mathsf{Decode}_{d_v}(c + d_u \cdot k \cdot n/8), d_v)$
3: $\hat{\mathbf{s}} := \mathsf{Decode}_{12}(sk)$
4: $m := \mathsf{Encode}_1(\mathsf{Compress}_q(v - \mathsf{NTT}^{-1}(\hat{\mathbf{s}}^T \circ \mathsf{NTT}(\mathbf{u})), 1))$ ▷ $m := \mathsf{Compress}_q(v - \mathbf{s}^T\mathbf{u}, 1))$
5: **return** m

Figure 4.6: Kyber.CPAPKE.Dec(sk, c) [18]

4.5 Kyber.CCAKEM

Kyber uses a variant of the Fujisaki–Okamoto (FO) transform [18] as depicted in figure 4.7 to level up the algorithm from INDCPA-secure to IND-CCA2-secure which is more secure than the previous one. The FO transform enables the creation of a KEM scheme with IND-CCA2 security starting from the previous IND-CPA-secure Kyber-PKE scheme.

The level of IND-CCA2 is suitable for real world application contexts. We will now see in more detail the three cryptographic algorithms present in this phase.

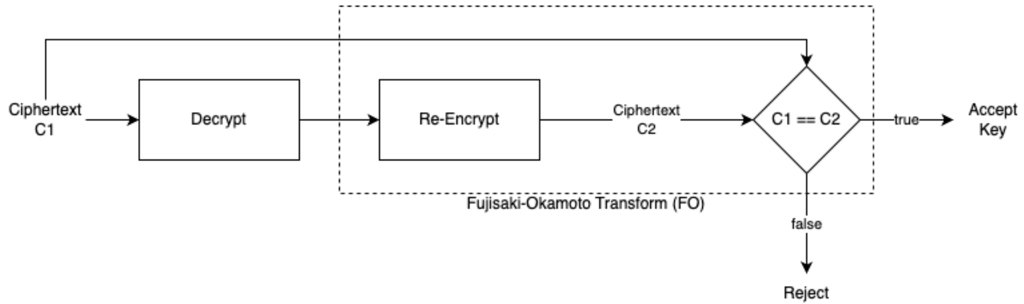


Figure 4.7: Fujisaki-Okamoto Transform (FO) for IND-CCA2-secure Decapsulation

4.5.1 Key Generation

The KeyGen() algorithm in figure 4.8 concatenates *sk*, *pk*, retrieved from the already described Kyber-CPAPKE.KeyGen(), with the output of the Hash function using *pk* as input, and with an additional secret *z* (line 3), which comes into play in case of rejection during the Decapsulation phase which we will explain shortly.

Algorithm 7 KYBER.CCAKEM.KeyGen()

Output: Public key $pk \in \mathcal{B}^{12 \cdot k \cdot n/8+32}$
Output: Secret key $sk \in \mathcal{B}^{24 \cdot k \cdot n/8+96}$
1: $z \leftarrow \mathcal{B}^{32}$
2: $(pk, sk') := \text{KYBER.CPAPKE.KeyGen}()$
3: $sk := (sk' \| pk \| \text{H}(pk) \| z)$
4: **return** (pk, sk)

Figure 4.8: Kyber.CCAKEM.KeyGen() [18]

4.5.2 Encapsulation (Client → Server)

In this algorithm depicted in figure 4.9 the input is the public key. Hash functions H and G (line 3) are involved in order to obtain: $\hat{K}$ which is the variable from which its computed the shared key K using the KDF, and r which is the random coin for the generation of the errors in the Kyber.CPAPKE encryption (line 4).

Algorithm 8 KYBER.CCAKEM.Enc(pk)

Input: Public key $pk \in \mathcal{B}^{12 \cdot k \cdot n/8+32}$
Output: Ciphertext $c \in \mathcal{B}^{d_u \cdot k \cdot n/8 + d_v \cdot n/8}$
Output: Shared key $K \in \mathcal{B}^*$
1: $m \leftarrow \mathcal{B}^{32}$
2: $m \leftarrow \text{H}(m)$ ▷ Do not send output of system RNG
3: $(\bar{K}, r) := \text{G}(m \| \text{H}(pk))$
4: $c := \text{KYBER.CPAPKE.Enc}(pk, m, r)$
5: $K := \text{KDF}(\bar{K} \| \text{H}(c))$
6: **return** (c, K)

Figure 4.9: Kyber.CCAKEM.Enc(pk) [18]

4.5.3 Decapsulation (Server → Client)

The Decapsulation function in figure 4.10 requires as input both the ciphertext and secret key. From secret key it is possible to extract pk, h and z (lines 1,2 and 3). In particular, for achieving the property of being indistinguishability under adaptive chosen ciphertext attack, the FO transformation is used in this algorithm. Once a ciphertext is decrypted, the corresponding plaintext is re-encrypted and compared with the received one. The purpose of Fujisaki-Okamoto is to add a level of information validation. More in detail, if the comparison between c' and c (line 7) yields a positive result, then the correct shared key K is derived and sent (line 8); otherwise, the ciphertext is deemed invalid and rejected by a random key which is generated from the previously sampled secret z (line 10). In this scenario, both parties possess distinct cryptographic keys, and any discrepancy is only detected in a subsequent communication phase, a concept referred to as implicit rejection. This strategy is designed to enhance resilience against misuse (e.g., absence of decapsulation outcome verification).

Algorithm 9 KYBER.CCAKEM.Dec(c, sk)

Input: Ciphertext $c \in \mathcal{B}^{d_u \cdot k \cdot n/8 + d_v \cdot n/8}$
Input: Secret key $sk \in \mathcal{B}^{24 \cdot k \cdot n/8 + 96}$
Output: Shared key $K \in \mathcal{B}^*$
1: $pk := sk + 12 \cdot k \cdot n/8$
2: $h := sk + 24 \cdot k \cdot n/8 + 32 \in \mathcal{B}^{32}$
3: $z := sk + 24 \cdot k \cdot n/8 + 64$
4: $m' :=$ KYBER.CPAPKE.Dec(sk, c)
5: $(\bar{K}', r') := \mathrm{G}(m' \| h)$
6: $c' :=$ KYBER.CPAPKE.Enc(pk, m', r')
7: **if** $c = c'$ **then**
8:     **return** $K := \mathsf{KDF}(\bar{K}' \| \mathrm{H}(c))$
9: **else**
10:     **return** $K := \mathsf{KDF}(z \| \mathrm{H}(c))$
11: **end if**
12: **return** K

Figure 4.10: Kyber.CCAKEM.Dec(c, sk) [18]

Chapter 5

Rust Programming Language

5.1 Introduction

The chapter starts with a quick explanation of the main weaknesses and vulnerabilities of a software written in C language because it mainly adopted for system programming and for embedded systems that is our case of study. The main reason for adopting this structure is to have an exhaustive picture of which are the points that have to be prevented. In C/C++ the correct security implementation and management is solely left as the responsibility of the programmer. In order to be able to face these limits of the language, the Rust model intervenes.

5.2 Preliminaries

Before moving on to explain Rust, we will introduce the main definitions that we will use in this chapter.

5.2.1 Vulnerability

The International Organization for Standardization (ISO) published the ISOIEC 27005 2008 [21] which is a standard document that provides guidelines for information security risk management. The ISOIEC 27005:2008 defines vulnerability as
"A weakness of an asset or group of assets that can be exploited by one or more threats, where an asset is anything that has value to the organization, its business operations, and their continuity, including information resources that support the organization's mission"
In order to enhance the ability of the organizations to recognize and respond to

cyber threats and vulnerabilities, the United States' National Cybersecurity (NFC) Federally Funded Research and Development Center (FFRDC) operated by the **MITRE** corporation publishes a list of Common Vulnerabilities and Exposures (**CVE**) [22].

5.2.2 Weakness

Weakness generally refers to a condition in a software or hardware that, could lead to the introduction of vulnerabilities; generally these conditions are errors (in software code, errors are also known as bugs), because one or more weaknesses can be exploited by an attacker to perform a malicious action. Errors are typically introduced by developers during development phase of the product. As in the vulnerability case, once again the **MITRE** corporation publishes a list of Common Weakness Enumeration (**CWE**).

5.3 CVE for Linux Kernel

We will study the case of the kernel of Linux to understand the importance of the responsibility of the programmer to don't introduce weaknesses into a software. Linux kernel is written in a special C programming language compatible with GCC compiler. In particular, C89 and C11 (since Linux 5.18 version) and assembly language were used in order to program the kernel. Figure 5.1 shows the number of vulnerabilities per year, recorded since 1999 for the Linux kernel.
As we can see, over about 24 years, the most common vulnerability types were:

- 44.1% of the total vulnerabilities were due to Denial of Service (**DoS**)
- 13.7% of the total, deriving from **Overflow** errors
- 4.8% due to **Memory Corruption**.
- 37.4% others.

Thus, about 20% of the known vulnerabilities are due to overflow and memory corruption circumstances, on the next chapter we will see that most of them can be completely prevented by the Rust compiler.

Table 5.1 shows the CVEs recorded for the year 2022 and filters only by vulnerabilities resulting from overflow errors and memory corruption. In this table, each vulnerability is associated with a CVE-ID and a CWE-ID, in particular this last parameter will be useful for us to better to understand the why of each mechanisms and concepts adopted by Rust programming language.

Year	# of Vulnerabilities	DoS	Code Execution	Overflow	Memory Corruption	Sql Injection	XSS	Directory Traversal	Http Response Splitting	Bypass something	Gain Information	Gain Privileges	CSRF	File Inclusion	# of exploits
1999	19	7		3						1		2			
2000	5	3										1			
2001	23	7								4		3			
2002	15	3		1						1	1				
2003	19	8		2						1	3	4			
2004	50	20	5	12							5	11			
2005	133	90	19	19	1					6	5	7			
2006	89	60	5	7	7			2		5	3	3			
2007	59	39	2	8						3	7	6			
2008	69	42	3	16	3					4	7	10			
2009	104	66	2	21	7					8	10	22			4
2010	118	62	3	16	7					8	31	14			5
2011	80	60	1	21	9					1	21	9			1
2012	114	83	3	24	10					6	19	11			
2013	186	99	6	38	13					11	57	25			7
2014	128	87	6	18	10					10	29	19			10
2015	79	52	5	13	4					10	10	14			
2016	215	153	5	36	18					12	34	51			1
2017	451	147	169	50	26					17	89	36			
2018	180	84	3	29	10					4	20	3			
2019	291	105	10	29	7			1		5	18	1			
2020	127	25	5	12	5			1		7	7	3			
2021	162	21	11	17	5			2		7	17	4			
2022	307	47	8	33	5					8	29	15			
2023	131	21	7	8	4					2	11	1			
Total	3154	1391	278	433	151			6		141	433	275			28
% Of All		44.1	8.8	13.7	4.8	0.0	0.0	0.2	0.0	4.5	13.7	8.7	0.0	0.0	

Figure 5.1: The list of all CVE recorded from 1999 to 2023. [23]

5.4 CWE for C language

Software developers use CWE for reporting weakness in software and discussing how to eliminate and/or mitigate them.
Table 5.2 submitted on 2008 and continuously updated, reports known weakness for a software written in C programming language. Using the CWE-IDs of the previous table 5.1, we can observe that 8/11 that means 73% of the overflow and memory corruption vulnerabilities that had been found in the Linux kernel derive from a well-known weakness for C programming language. The CWE-IDs 119, 120, 121, 362, 416, 476, 787 and 843 from table 5.1 are found on table 5.2 and some of them are listed below because are relevant in order to understand the importance of some Rust concepts and rules.
The main observation of the C language is that the responsibility of not introducing weaknesses to the code is left entirely to the programmer.

- Use after free (**CWE-416**): Referencing memory after it has been deallocated can lead to program crashes or unintended code execution

Table 5.1: Overflow and Memory Corruption vulnerabilities for the year 2022

CVE ID	CWE ID	Vulnerability Type(s)	Publish Date	Update Date	Score
CVE-2022-0435	787	Overflow	2022-03-25	2023-02-14	9.0
CVE-2021-4157	119	Overflow	2022-03-25	2023-01-17	7.4
CVE-2022-0500	119	Overflow	2022-03-25	2023-03-01	7.2
CVE-2022-0185	190	Overflow	2022-02-11	2023-02-12	7.2
CVE-2022-0998	190	Overflow	2022-03-30	2023-03-01	7.2
CVE-2022-1116	190	Overflow Mem. Corr.	2022-05-17	2022-10-19	7.2
CVE-2022-1116	190	Overflow Mem. Corr.	2022-05-17	2022-10-19	7.2
CVE-2022-34918	843	Overflow	2022-07-04	2023-05-16	7.2
CVE-2021-3428	190	DoS Overflow	2022-03-04	2022-03-11	4.9
CVE-2022-25258	476	Mem. Corr.	2022-02-16	2022-12-07	4.9
CVE-2022-26490	120	Overflow	2022-03-06	2023-01-20	4.6
CVE-2022-32981	120	Overflow	2022-06-10	2022-06-27	4.6
CVE-2022-27666	787	Overflow	2022-03-23	2023-02-01	4.6
CVE-2022-2078	121	DoS Overflow	2022-06-30	2022-10-26	2.1
CVE-2022-2964	119	Overflow	2022-09-09	2023-01-20	0.0
CVE-2022-3435	119	Overflow	2022-10-08	2023-03-01	0.0
CVE-2022-3541	119	Overflow	2022-10-17	2023-02-02	0.0
CVE-2022-3545	119	Overflow	2022-10-17	2023-05-12	0.0
CVE-2022-3564	119	Overflow	2022-10-17	2023-02-23	0.0
CVE-2022-3565	119	Overflow	2022-10-17	2023-02-06	0.0
CVE-2022-3625	119	Overflow	2022-10-21	2023-02-10	0.0
CVE-2022-3635	119	Overflow	2022-10-21	2023-05-26	0.0
CVE-2022-3636	119	Overflow	2022-10-21	2023-02-23	0.0
CVE-2022-3640	119	Overflow	2022-10-21	2023-04-11	0.0
CVE-2022-3649	119	Overflow	2022-10-21	2023-05-26	0.0
CVE-2022-3077	120	Overflow	2022-09-09	2022-09-15	0.0
CVE-2022-36402	190	DoS Overflow +Priv	2022-09-16	2022-09-20	0.0
CVE-2022-39842	190	Overflow Bypass	2022-09-05	2023-03-01	0.0
CVE-2022-45869	362	DoS Mem. Corr.	2022-11-30	2022-12-05	0.0
CVE-2021-3759	400	DoS Overflow	2022-08-23	2023-03-01	0.0
CVE-2022-1976	416	Mem. Corr.	2022-08-31	2023-02-14	0.0
CVE-2022-2938	416	Mem. Corr.	2022-08-23	2023-01-20	0.0
CVE-2022-43945	770	Overflow	2022-11-04	2023-03-08	0.0
CVE-2022-2991	787	Exec Code Overflow	2022-08-25	2022-08-30	0.0
CVE-2022-41674	787	Overflow	2022-10-14	2023-03-01	0.0
CVE-2022-47518	787	Overflow	2022-12-18	2023-05-12	0.0
CVE-2022-47521	787	Overflow	2022-12-18	2023-04-11	0.0
CVE-2022-47942	787	Overflow	2022-12-23	2023-05-16	0.0

```
1  #include <stdio.h>
2  #include <unistd.h>
3  #define BUFSIZER1 512
4  #define BUFSIZER2 ((BUFSIZER1/2) - 8)
5
```

```
int main(int argc, char **argv) {
    char *buf1R1;
    char *buf2R1;
    char *buf2R2;
    char *buf3R2;
    buf1R1 = (char *) malloc(BUFSIZER1);
    buf2R1 = (char *) malloc(BUFSIZER1);
    free(buf2R1);
    buf2R2 = (char *) malloc(BUFSIZER2);
    buf3R2 = (char *) malloc(BUFSIZER2);
    strncpy(buf2R1, argv[1], BUFSIZER1-1);
    free(buf1R1);
    free(buf2R2);
    free(buf3R2);
}
```

- Double free (**CWE-415**): The product invokes free() twice on the same memory address, which can result in unintended modifications to memory locations

```
char* ptr = (char*)malloc (SIZE);
...
if (abrt) {
free(ptr);
}
...
free(ptr);
```

- Missing Release of Memory after Effective Lifetime (**CWE-401**): The product does not sufficiently track and release allocated memory after it has been used, which slowly consumes remaining memory.

```
char* getBlock(int fd) {
    char* buf = (char*) malloc(BLOCK_SIZE);
    if (!buf) {
    return NULL;
    }
    if (read(fd, buf, BLOCK_SIZE) != BLOCK_SIZE) {
```

```
    return NULL;
    }
    return buf;
}
```

- NULL Pointer Dereference (**CWE-476**): A NULL pointer dereference happens when the application attempts to access or use a pointer that it assumes is valid, but it is actually NULL, typically causing a crash or exit.

```
void host_lookup(char *user_supplied_addr){
    struct hostent *hp;
    in_addr_t *addr;
    char hostname[64];
    in_addr_t inet_addr(const char *cp);

    /*routine that ensures user_supplied_addr
    is in the right format for conversion */

    validate_addr_form(user_supplied_addr);
    addr = inet_addr(user_supplied_addr);
    hp = gethostbyaddr( addr, sizeof(struct in_addr), AF_INET);
    strcpy(hostname, hp->h_name);
}
```

Table 5.2: Weaknesses in Software Written in C, submission date 2008-04-11 [24]. Some of them are highlighted in gray and are prevented at compile-time by the Rust compiler

CWE ID	Name	Weakness Abstraction	Status
14	Compiler Removal of Code to Clear Buffers	Variant	Draft
119	Improper Restriction of Operations within the Bounds of a Memory Buffer	Class	Stable
120	Buffer Copy without Checking Size of Input ('Classic Buffer Overflow')	Base	Incomplete
121	Stack-based Buffer Overflow	Variant	Draft
122	Heap-based Buffer Overflow	Variant	Draft
123	Write-what-where Condition	Base	Draft
124	Buffer Underwrite ('Buffer Underflow')	Base	Incomplete
125	Out-of-bounds Read	Base	Draft
126	Buffer Over-read	Variant	Draft
127	Buffer Under-read	Variant	Draft
128	Wrap-around Error	Base	Incomplete
129	Improper Validation of Array Index	Variant	Draft
130	Improper Handling of Length Parameter Inconsistency	Base	Incomplete
131	Incorrect Calculation of Buffer Size	Base	Draft
134	Use of Externally-Controlled Format String	Base	Draft
135	Incorrect Calculation of Multi-Byte String Length	Base	Draft
170	Improper Null Termination	Base	Incomplete
188	Reliance on Data/Memory Layout	Base	Draft
191	Integer Underflow (Wrap or Wraparound)	Base	Draft
192	Integer Coercion Error	Variant	Incomplete
194	Unexpected Sign Extension	Variant	Incomplete
195	Signed to Unsigned Conversion Error	Variant	Draft
196	Unsigned to Signed Conversion Error	Variant	Draft
197	Numeric Truncation Error	Base	Incomplete
242	Use of Inherently Dangerous Function	Base	Draft
243	Creation of chroot Jail Without Changing Working Directory	Variant	Draft
244	Improper Clearing of Heap Memory Before Release ('Heap Inspection')	Variant	Draft
362	Concurrent Execution using Shared Resource with Improper Synchronization ('Race Condition')	Class	Draft
364	Signal Handler Race Condition	Base	Incomplete
366	Race Condition within a Thread	Base	Draft
374	Passing Mutable Objects to an Untrusted Method	Base	Draft
375	Returning a Mutable Object to an Untrusted Caller	Base	Draft
401	Missing Release of Memory after Effective Lifetime	Variant	Draft
415	Double Free	Variant	Draft
416	Use After Free	Variant	Stable
457	Use of Uninitialized Variable	Variant	Draft
460	Improper Cleanup on Thrown Exception	Base	Draft

CWE ID	Name	Weakness Abstraction	Status
462	Duplicate Key in Associative List (Alist)	Variant	Incomplete
463	Deletion of Data Structure Sentinel	Base	Incomplete
464	Addition of Data Structure Sentinel	Base	Incomplete
466	Return of Pointer Value Outside of Expected Range	Base	Draft
467	Use of sizeof() on a Pointer Type	Variant	Draft
468	Incorrect Pointer Scaling	Base	Incomplete
469	Use of Pointer Subtraction to Determine Size	Base	Draft
474	Use of Function with Inconsistent Implementations	Base	Draft
476	NULL Pointer Dereference	Base	Stable
478	Missing Default Case in Multiple Condition Expression	Base	Draft
479	Signal Handler Use of a Non-reentrant Function	Variant	Draft
480	Use of Incorrect Operator	Base	Draft
481	Assigning instead of Comparing	Variant	Draft
482	Comparing instead of Assigning	Variant	Draft
483	Incorrect Block Delimitation	Base	Draft
484	Omitted Break Statement in Switch	Base	Draft
495	Private Data Structure Returned From A Public Method	Variant	Draft
496	Public Data Assigned to Private Array-Typed Field	Variant	Incomplete
558	Use of getlogin() in Multithreaded Application	Variant	Draft
560	Use of umask() with chmod-style Argument	Variant	Draft
562	Return of Stack Variable Address	Base	Draft
587	Assignment of a Fixed Address to a Pointer	Variant	Draft
676	Use of Potentially Dangerous Function	Base	Draft
685	Function Call With Incorrect Number of Arguments	Variant	Draft
688	Function Call With Incorrect Variable or Reference as Argument	Variant	Draft
689	Permission Race Condition During Resource Copy	Compound	Draft
690	Unchecked Return Value to NULL Pointer Dereference	Compound	Draft
704	Incorrect Type Conversion or Cast	Class	Incomplete
733	Compiler Optimization Removal or Modification of Security-critical Code	Base	Incomplete
762	Mismatched Memory Management Routines	Variant	Incomplete
781	Improper Address Validation in IOCTL with METHOD_NEITHER I/O Control Code	Variant	Draft
782	Exposed IOCTL with Insufficient Access Control	Variant	Draft
783	Operator Precedence Logic Error	Base	Draft
785	Use of Path Manipulation Function without Maximum-sized Buffer	Variant	Incomplete
787	Out-of-bounds Write	Base	Draft

CWE ID	Name	Weakness Abstraction	Status
789	Memory Allocation with Excessive Size Value	Variant	Draft
805	Buffer Access with Incorrect Length Value	Base	Incomplete
806	Buffer Access Using Size of Source Buffer	Variant	Incomplete
839	Numeric Range Comparison Without Minimum Check	Base	Incomplete
843	Access of Resource Using Incompatible Type ('Type Confusion')	Base	Incomplete
910	Use of Expired File Descriptor	Base	Incomplete
911	Improper Update of Reference Count	Base	Incomplete
1325	Improperly Controlled Sequential Memory Allocation	Base	Incomplete
1335	Incorrect Bitwise Shift of Integer	Base	Draft
1341	Multiple Releases of Same Resource or Handle	Base	Incomplete

5.5 Rust Overview

Rust is a multi-paradigm programming language focuses on two main aspects.

- Safety: 70% of the security issues that the Microsoft Security Response Center (MSRC) assigns a CVE to are memory safety issues [25]. As already described, about 20% of the Linux Kernel vulnerabilities are due to overflow and memory corruption cases. The Rust's Ownership model guarantees memory safety and thread-safety through the control that the Borrow Checker carries out; therefore, a lot of security issues can be eliminated at compile-time.

- Performance: Rust has not a garbage collector so it do not requires additional cpu cycles to handle this function. Can be used on embedded systems and can be integrated with other languages (i.e C).

It was developed as a personal project by Graydon Hoare, a Mozilla researcher employee. On 2009, the company expressed interest and announced the language in the Mozilla Summit 2010. The Rust compiler was initially written in OCaml, but the project shifted to a self-hosting compiler based on LLVM written in Rust. In 2015 was released the 1.0 version of the language.

5.5.1 Safety - Pointers and Memory

Rust enforces Resource Acquisition Is Initialization (RAII) that is a programmatic idiom that describes very useful behaviors with important benefits. This technique allows whenever an object goes out of scope (even in case of errors) to be correctly released with the destruction of the resources acquired.

Borrow Checker

The Borrow Checker is a compiler module whose function is to verify that all references respect the **single writer or multiple readers rule**. In the example below, two common cases are explained: the not allowed re-assign and the forbidden immutable borrow (the original variable is currently borrowed as mutable).

```
1  fn borrow() {
2      let mut a = 5;
3      let b = &a;
4      println!("{}", *b);
5      a = a+1; // Cannot assign to `a` because it is borrowed
6      println!("{}", *b);
7  }
8
9  fn mutable_borrow() {
10     let mut a = 7;
11     let b = &mut a;
12     println!("{}", a); // cannot borrow `a` as immutable
13                        // because it is also borrowed as mutable
14     *b = *b+1;
15     println!("{}", *b);
16 }
```

References in Rust can never be NULL or point to addresses whose value has already been released or has never been initialized, in order to avoid a potential NULL pointer dereference (**CWE-476**). This is a significant difference from classic pointers in C/C++.

Unsafe block

The compiler always checks the ownership and lifetime of the variables, and ensures that accesses that are legitimate. If the programmer finds himself in a condition that does not satisfy these criteria, he must be forced to use an **unsafe{...}** block that surrounds the unsafe code block.

5.5.2 Syntax

Variables and types

The Rust primitive types of the language are:

- the unit, that means a tuple with no elements inside (corresponding to void in C/C++): **()**
- unsigned integers: **u8, u32, u64, u128, usize**
- signed integers: **i8, i32, i64, i128, isize**
- floating point numbers: **u32, f64**
- booleans: **bool**
- characters: **char**
- strings: **String, &str**

For the sake of synthesis, we prefer to just mention that there exist other important Rust data types and structures: Collections, Iterators, Smart-Pointers (very important for thread-safety context therefore not relevant for our case) etc..

Rust preferred to use **let** for the **immutable** assignment of a value to a variable. The programmer is therefore forced to understand if in the future he will need to **mutate** the value, so declaring it as mutable with the **mut** label. The type can be declared by the programmer when declaring the variable or automatically **inferred** by the **type inference engine**. The variable is statically associated with the declared or inferred type, and this means that the variable will have only that type for the entire duration of the program.

```rust
fn main() {
    let a: u32 = 17;
    // a = 3 -> error
    let mut b = a; // b will have the same type as a
                   // and the value of b can be reassigned
    b = 3;
}
```

5.5.3 Ownership

Rust's Ownership concept means that any data introduced into the program is owned by exactly one variable (called the **owner**) that owns also the resource and is responsible for the correct deallocation. This is the reason why there isn't a garbage collector. The concept of possession implies responsibility for releasing the variable allocated memory. This mechanism occurs when the owner goes out of its syntactic scope or when it is directly assigned a new value to it. Applying this

rules ensures the removal of many undefined behaviors caused by having pointers pointing to no longer valid address spaces. Therefore, Rust's idea of ownership prevents the programmer to create dangling pointers on the program.

```rust
fn main() {
    let mut a = Vec::new();
    a.push(1);
    a.push(2);
}
/*at this point, the array a goes out of its syntactic scope
(after exiting of the previous function), the variable a is
responsible of releasing the resources it owns, that is the
dynamic array allocated on the heap. */
```

It is possible to change the ownership through the mechanism of **movement**, that means changing the owner of the data. This mechanism is the default behavior in Rust.

```rust
fn main() {
    let mut a = String::from("Test");
    // use of the varible a
    let b = a;
    /*variable a is still alive but its internal value is marked
    as no longer accessible and consequently the ex owner cannot read
    it because it has been moved (has changed owner) to variable b*/

    //use of the variable b
}
```

The movement action occurs even if we pass a data as a parameter of a function as showed below. We are therefore transferring the property and the original owner will no longer own the property.

```rust
fn main() {
    let mut a = vec![1,2,3];
    handle(a);
    // variable a is no longer usable here because was moved
    // in the previous function
    println!("{:?}", a); //compile error
```

```rust
}

fn handle(a) {
    println!("{:?}", a); // a is correctly displayed
}
```

5.5.4 Borrowing

In order to pass a value as an input parameter to a function without losing ownership (due to the default movement mechanism) we can pass the reference to the value owned by the original variable. A simple reference, therefore, is a read-only pointer to a block of memory held by another variable that allows you to access a value without transferring ownership. It is necessary to indicate that primitives types (i.e integer,float..) and other types that implement the Clone trait can be moved to other functions without loosing the ownership because they are copied and not moved.

As long as the reference is accessible, it is not possible to change the value either by the reference or by the variable (the real owner) that owns the value, but it is possible to create further simple references from the original data or from other references to it.

```rust
fn main() {
    let v = vec![10,20,30];
    print_vector(&v);
    println!("{}", v[0]);
}

fn print_vector(x: &Vec<i32>) {
    println!("{}", x);
}
```

5.5.5 Mutability

When there is a need to obtain a mutable borrow, because we need to modify the referenced value, we can use mutable references. In particular, from a mutable variable that currently owns a value, only one mutable reference can be declared at a time, and there can be no simple references to the original variable.

```rust
fn add_two(r: &mut i32) {
    *r += 2;
}

fn main() {
    let mut a = 5;
    add_two(&mut a);
    println!("{}", a);
}
```

5.5.6 Lifetimes

In order to avoid the phenomenon of "Use After Free" (**CWE-416**), the Borrow Checker uses Lifetimes to keep track of how long references are valid for. In particular, this check consists in verifying that each reference access takes place in a time interval included in the value lifetime. To be legitimate, the reference must last less than the lifetime of the pointed variable. The C/C++ compiler does not do this check and consequently fails to detect the presence of any dangling pointers.

Lifetimes annotation syntax

The notation of a lifetime parameter starts with an apostrophe '.

```rust
&i32         /* simple read-only reference */
&'a i32      /* reference with an explicit lifetime */
&'a mut i32  /* mutable reference with an explicit lifetime */
&'static str /* reference with lifetime valid for the entire
                duration of the program since the character sequence
                is allocated by the compiler in the constants section
                and therefore is never released */
```

In cases where there is no ambiguity and therefore the compiler can understand the lifetime, and it is not necessary to make it explicit. Indeed, there is a procedure called **lifetime inference** in which the compiler adds to each borrow an interval of time, which represents the duration between starting point of the borrow and its last use. Therefore, in the below example, the lifetime inference process computed by the compiler proceeds to add an interval of existence for the variable b (from line 5 to 7) and an interval of validity for variable a (from line 5 to 10).

```rust
fn main() {
    let a;

    {
        let b = 2;      // -+-- 'b -+-- 'a
        a = &b;         //  |       |any reference to b cannot exceed this period
                        //  |       |
    }                   // -+       |b is dropped after exit the scope here
                        //          |
    println!("{}", a); //          |borrowed value (&b) does not live enough
}                       // ---------+
```

Lifetimes in function signatures

If a function receives as input only one reference, the compiler, in case of non-ambiguity, automatically translates the signature of the function into the one that can be observed below. This process is called **lifetime elision**.

```rust
fn handle() { ... } => lifetime elision => fn handle<'a>(i: &'a i64) { ... }
```

If a function receives more than one reference and the compiler finds itself within a situation of ambiguity, it may be necessary to indicate whether the lifetime is constrained to the shortest of all parameters (line 1) or if they are completely disjoint (line 2).

```rust
fn handle<'a>(i1: &'a i64, i2: &'a i64) { ... }
fn handle<'a, 'b>(i1: &'a i64, i2: &'b i64) { ... }
```

A typical case of ambiguity occurs when the compiler fails automatically to figure out the correct lifetime of functions that return a reference to one of the function's input parameters as showed below. To resolve this ambiguity it is necessary that the returned reference (&'b i32) arises in some way from the second parameter (from which it derives its life span) so that the result also has the same duration.

```rust
fn handle<'a, 'b>(i1: &'a i64, i2: &'b i64) -> &'b i32 { return &i2.x; }
```

5.5.7 Polymorphism

Polymorphism let us to have the idea that a variable can take multiple forms, and it occurs when we have many classes that are related to each other by inheritance. The main disadvantage of polymorphism in many programming languages is that the system has to determine which process or variable need to be invoked and this decision is taken at run time, and therefore the performance of the program can **decrease**.
Rust doesn't have the concept of inheritance because types are **not** organized in a hierarchical model. Something similar to the polymorphism concept can be achieved using generic types and traits explained below.

Generics

Rust generics equivalent to template programming in C++ use the concept of meta-programming whereby a generic data type can be expressed through the use of a meta-variable (i.e **T**). The parametric polymorphism approach allows programmers to write functions leaving a variable type to be partially specified (Generic) in order to allow the same function to be applied to different variable types reducing duplicate code sections. To handle the case of generic types many languages like Java only know the true data type at run-time. This has a performance impact.

Rust, instead, uses the **monomorphization** process in order to know the concrete type of all generic types used by the program, and this process is done at compile-time so before the code is executed. To do this, the compiler generates several copies of each concrete type that the program actually needs. For example, if both Vec<i8> and Vec<u128> types are used in the code then the generated binary will contain two different copies of Vec, one for the case Vec<i8> and the other for the case Vec<u128>. This process has the disadvantage of having a longer compilation time since multiple copies must be generated, so more quantity of code, which leads to a greater size of the compiled binary file but the great advantage is that this process allows of having no runtime costs in order to use the correct concrete type, therefore better performance.

```
fn handle<T,R>(i1: &Data, i2: T, i3: R) -> Results
    where T: Serialize + Deserialize
          R: Read
{ ... }
```

Traits

In order to achieve a similar behavior of Java/C# interfaces or pure abstract classes in C++, we can use Traits. The trait is used to allow a data type to have certain behaviors (functionalities). In the example below, we see that the two data types (Candidate and Finalist) can both perform the math actions of summing items and calculating the average. Moreover, if we don't have dynamic references (&dyn) we can invoke a trait-related methods on a value without incurring any additional costs in terms of CPU cycles. However, this does not happen in C++ or Java where a penalty must be paid in order to use interface methods.

```
trait Math {
    fn sum() -> i32;
    fn average() -> f32;
}

struct Candidate;
struct Finalist;

impl Math for Candidate {
    fn sum() -> i32 {
        // compute and return the sum
    }
    fn average() -> f32 {
        // compute and return the average
    }
}

impl Math for Finalist {
    fn sum() -> i32 {
        // compute and return the sum
    }
    fn average() -> f32 {
        // compute and return the average
    }
}
```

5.6 The no_std environment

When we are dealing with embedded programming, is very important to classify two different environments:

- **std**: the standard library can be used in general or special (hosted) purpose systems. In these environments exist an interface that exposes primitives which allow the interaction with many components like peripherals, networking, memory management, threads, file system, etc.

- **no_std**: This is the environment that we will use for our analisys, because it can be used for bare metal environments (i.e microcontroller units), in which there is no OS that can provide a software layer to interact with kernel calls. This crate-level attribute prevent Rust from loading components that use the standard library, and will link only to the core-crate instead of the std-crate.

5.7 Motivation of adopting Rust for the project

- NIST included [26] Rust in its list of safer languages, and declares: *Rust has an ownership model that guarantees both memory safety and thread safety, at compile-time, without requiring a garbage collector. This allows users to write high-performance code while eliminating many bug classes. Though Rust does have an unsafe mode, its use is explicit, and only a narrow scope of actions is allowed. (14 Mar 2023)*

- In December 2022 after the release of Linux 6.1, it became the second high-level language to be supported in the development of the Linux kernel.

- It is very fast compared to other system programming languages and for being a trade-off between performance and safety.

Chapter 6

Kyber library written in Rust on a ARM Cortex-M4

6.1 The STM32F303VCT6 Board

6.1.1 Development environment for STM32

The STM manufacturers guarantee a complete support for the development lifecycle of a STM32 microcontroller that uses C/C++ languages for developing. The STM team provides software tools such as:

- STM32CubeIDE: that is a integrated development environment
- STM32CubeMonitor: a monitoring tool to test at run-time
- Arm Keil MDK: a c/c++ compiler, debugger and IDE for stm32
- Other tools from other suppliers

This is not the case for Rust programming language, which only has a growing but still young community of developers that try to create a compatible environment. See chapter 5.1 for more details on Rust language.

6.1.2 Our selection

To keep up with the progress made by the Rust community that guarantees and documents the compatibility of hardware and to avoid encountering unknown errors, we preferred to use the STM32F303VCT6 board (also known as *STM32F3DISCOVERY*). Indeed, it have been chosen by the community, and then its usage is so very well documented in the manual called *Rust Embedded Book* [27].

This microcontroller has the following main features:

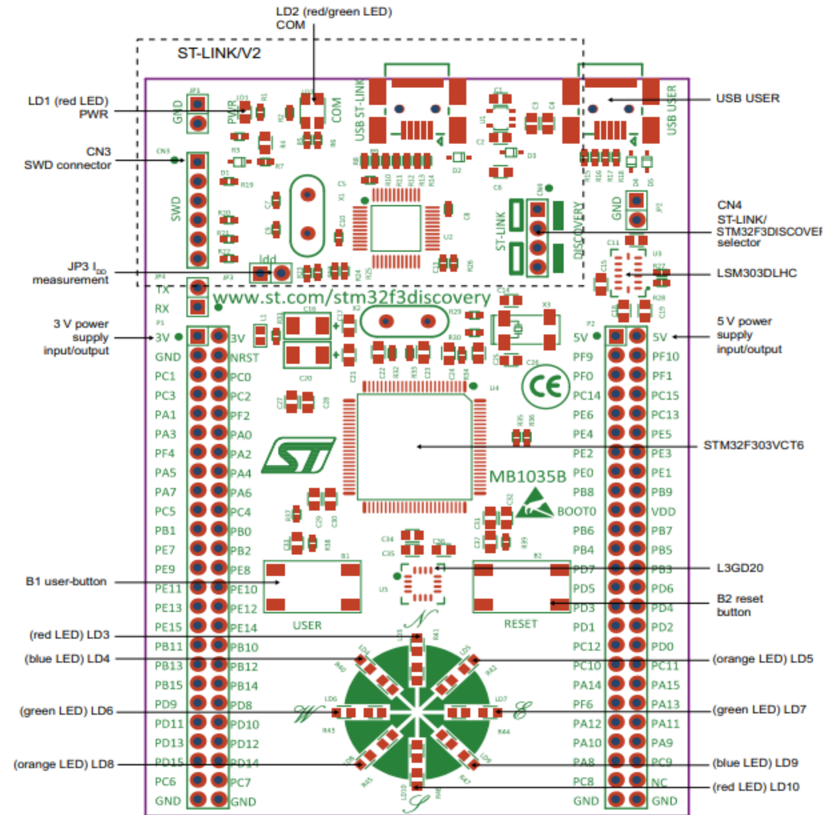


Figure 6.1: STM32f303VCT6 Layout (top view) [28]

- Arm Cortex M4 32bit Single-core CPU (maximum clock frequency of 72 MHz)
- 40 KB of SRAM with HW parity check implemented on the first 16KB and 8KB of Core-Coupled-Memory (CCM) with HW parity check.
- 256 KB of Flash memory
- Single-cycle multiplication
- MPU (Memory Protection Unit)
- 4 to 32MHz crystal oscillator, 32kHz oscillator for RTC with calibration, internal 8MHZ RC with 16 PLL option, internal 40kHz oscillator
- 87 I/Os mappable on external interrupt vectors
- A variety of integrated peripherals such as timers, I2C, SPI and USART.
- 12-channel DMA controller

- 4 ADCs

6.1.3 Limitations of the board

The board doesn't provide a random number generator unit in order to produce a non-deterministic random numbers.
For the tests we will use a function that is a pseudo-random number generation algorithm since it does not use physical phenomena to generate entropy. We will dedicate an entire chapter to discuss aspects related to randomness (see Chapter 7). In order to guarantee security to the users of the library, an external RNG peripheral should be adopted, and it should satisfy the NIST's *Recommendation for Random Bit Generator (RBG) Constructions* document [29].

6.2 State of Art of Kyber on a ARM Cortex-M4

6.2.1 PQClean

Apart from the official C reference code of Kyber, the PQClean project [30] is another widely used library, because satisfies a number of basic code quality requirements and contains a collection of highly-tested and high-confidence implementations of 17 NIST finalist post-quantum schemes. The strength of this project is the environment created through the use of automatic tests. These tests are run periodically on the master branch and on each commit and pull requests, and they are tested on different systems architectures and operating systems.

Flaw	KEMs	Sigs	Flaw	KEMs	Sigs		Test
Memory safety ◇	3	4	Endianness assumptions †	7	2	∗	Compilation test
Signed integer overflow ⋆, ◇, **†**	3	1	Platform-specific behavior ♣, ±, †, **†**	4	0	♠	`Makefile` checks
Alignment assumptions ⋆, ♣, ◇	4	4	Variable-Length Arrays ⋆	4	1	♣	Functional tests
Other Undefined Behavior ⋆, ♣	1	1	Compiler extensions ⋆	5	2	†	Test vectors
Dead code ⋆, ♠	3	4	Integer sizes ◇, ⋆, †	6	3	◇	Sanitizers
Global state	2	1	Non-constant time =	4	0	±	Signedness of `char`
Licensing unclear ©	3	1				=	Timing-suspicious ops.
						†	`clang-tidy`
						©	License file

Figure 6.2: List of known flaws. On Each flaw it is specified which tests might have detected them, from [30]

As we can see in figure 6.2, there are known flaws reported by PQClean project regarding memory safety, signed integer overflow, undefined behaviors, variable-length arrays, etc. These problems are dangerous in order to preserve security and can be avoided by using the Rust model as also PQClean declares [30]: *"Rust could perhaps have stood in as a lowlevel language that simply does not allow most of the problems that our testing system was designed to catch"*. Moreover,

contributors wonder whether is the right decision to adopt C as the principal project programming language: *"More controversially, we question if C is a suitable programming language for reference implementations, especially if the main goal is clarity of the implementation. While as of now there seems to be no consensus on which alternative should be used, standardization entities should revisit this on a regular basis"* [30].

6.2.2 pqm4

The *pqm4* [31] is a project specifically suited for the ARM Cortex-M4 embedded systems. This project defines as **clean** type a code version that integrates the PQClean's repo. There is also an implementation optimized for the Cortex-M4 which is called **m4** which is specially adapted to work with the default micro-controller *STM32F4DISCOVERY*. We only considered benchmarks of the pqm4's **clean** version.

6.3 Selected Rust Kyber Library for the board

We decide to analize the open source `https://github.com/Argyle-Software/kyber` library (Apache 2.0 - MIT). This library is the translation of the official reference code of Kyber library from C to Rust language. Then, this library takes the Rust language advantages such as performance and memory safety.
This library doesn't contain any type of binding with the C counterpart and no unsafe blocks are written. This is very useful in order to avoid any kind of potential exposures.
Finally, is suitable for our scope because it is intended to be compatible with embedded systems (since it is no_std compatible).

6.4 Performance Evaluation

Performance tests were carried out following the same configurations adopted by the *pqm4* framework to create benchmarks [32]. In particular, *pqm4* uses the STM32F407DISCOVERY [33] board (196 KB of memory and 1 MB flash ROM) and all clock cycles were measured at 24MHz of core frequency, to avoid wait cycles due to the speed of the memory controller [31].

6.4.1 Clock Cycles

A clock cycle (or clock tick) is the time between two pulses. Pulses are rising edges of a repetitive clock signal. From clock cycles is possible to determine the speed of

a CPU. This metric is called clock frequency and is measured in Hz,

$$f_c = \frac{1}{T_c}$$

where T_c is the number of clock cycles. The more pulses per second, the faster the computer processor can process information (i.e fetch, execute instructions).

6.4.2 Clock Cycles Measurement Method

Two approaches have been used to measure the clock cycles elapsed to complete the execution of a *KEM* method. Moreover, a comparison between these two measure methods and the pqm4 benchmarks are discussed on chapter 8.

SysTick

SysTick is a CPU's peripheral system timer that is used to measure elapsed cycle counts. This count is accurate up to a 24-bit maximum number of clock cycles. We choose as reload value (is the seed value when the timer expires) to be 24000000 / 2 = 12000000. We have decided this value so that it can be represented by the 24-bit SYST_RVR register available. When peripheral counts down from the reload value to zero it reloads to this value chosen and repeat the entire process again. This is the cpu and timer configuration:

```
1  let mut dp = pac::Peripherals::take().unwrap();
2  let mut flash = dp.FLASH.constrain();
3  let mut rcc = dp.RCC.constrain();
4  let clocks = rcc
5      .cfgr
6      .use_hse(8.MHz())
7      .sysclk(24.MHz())
8      .freeze(&mut flash.acr);
9  let core_periphs = cortex_m::Peripherals::take().unwrap();
10 let mut syst = core_periphs.SYST;
11 syst.set_clock_source(SystClkSource::Core);
12 syst.set_reload(clocks.sysclk().0 / fraction - 1);
13 syst.clear_current();
14 syst.enable_counter();
15 syst.enable_interrupt();
```

For obtained a *rng* variable we create a CustomRng structure that implements the RngCore and CryptoRng traits to be compliant with the library constraints. In

particular, we use the *fill_bytes()* function in order to fill random bytes in a fast way.

```
#[derive(Clone, Debug)]
use rand_core::{RngCore, CryptoRng, Error,impls};

pub struct CustomRng(u64);

impl RngCore for CustomRng {
    fn next_u32(&mut self) -> u32 {
        self.next_u64() as u32
    }

    fn next_u64(&mut self) -> u64 {
        self.0 += 1;
        self.0
    }

    fn fill_bytes(&mut self, dest: &mut [u8]) {
        impls::fill_bytes_via_next(self, dest)
    }

    fn try_fill_bytes(&mut self, dest: &mut [u8]) -> Result<(), Error> {
        Ok(self.fill_bytes(dest))
    }
}

impl CryptoRng for CustomRng {}

let mut rng = CustomRng(76187368 as u64);
```

The block below is used on each measurement and shows the logic to:

1. define the number of ticks starting from the initial and final instants read from the SYST_CVR register
2. how the minimum, maximum and sum are obtained (the last one is useful for averaging at the end)

```
let ticks = if inst0 > inst1 {
    inst0 - inst1 - 2
```

```
3 }else {
4     inst0 + (12000000 - inst1)
5 };
6 if min >= ticks {
7     min = ticks;
8 }
9 if max <= ticks {
10     max = ticks;
11 }
12 sum += ticks;
```

In the next code sections, we show how the initial and final clock cycles of the main *kem* functions are read.

```
1 /// KeyPair
2 let mut pk = [0u8; KYBER_PUBLICKEYBYTES];
3 let mut sk = [0u8; KYBER_SECRETKEYBYTES];
4 let bufs = Some(([1u8; 32].as_slice(), [255u8; 32].as_slice()));
5 let inst0 = syst.cvr.read();
6 crypto_kem_keypair(&mut pk, &mut sk, &mut rng, bufs);
7 let inst1 = syst.cvr.read();
```

For the sake of simplicity, *pk* (public key), *ct* (ciphertext) and *ss* (shared secret) vectors are mocked and obtained from the output of the KAT. We place this long vectors on the Appendix.

```
1 /// Encapsulate
2 let mut ct = [0u8; KYBER_CIPHERTEXTBYTES];
3 let mut ss = [0u8; KYBER_SSBYTES];
4 let encap_buf = Some([255u8; 32].as_slice());
5 let inst0 = syst.cvr.read();
6 crypto_kem_enc(&mut ct, &mut ss, &pk, &mut rng, encap_buf);
7 let inst1 = syst.cvr.read();
```

```
1 /// Decapsulate
2 let mut ss = [0u8; KYBER_SSBYTES];
3 let inst0 = syst.cvr.read();
4 res = match crypto_kem_dec(&mut ss, &ct, &sk) {
```

```
    Ok(_) => ss,
    Err(_) => [0u8;32]
};
let inst1 = syst.cvr.read();
```

Data Watchpoint and Trace (DWT) Unit

The DWT unit counts the execution cycles. This unit contains 4 counters. In particular, for our work is relevant the read-only clock cycle counter register called CYCCNT. This register increments on each clock cycle when the processor is not halted in debug state. The code section below show the CPU configuration and the counter setup.

```
let mut dp = pac::Peripherals::take().unwrap();
let mut flash = dp.FLASH.constrain();
let mut rcc = dp.RCC.constrain();
rcc
    .cfgr
    .use_hse(8.MHz())
    .sysclk(24.MHz())
    .freeze(&mut flash.acr);

let mut peripherals = Peripherals::take().unwrap();
peripherals.DWT.enable_cycle_counter();
```

The *op_cyccnt_diff()* macro below was written in order to restricts the kinds of memory re-ordering that the compiler normally does.

```
macro_rules! op_cyccnt_diff {
    ( $( $x:expr )* ) => {
        {
            compiler_fence(Ordering::Acquire);
            let before = DWT::cycle_count();
            $(
                $x;
            )*
            let after = DWT::cycle_count();
            compiler_fence(Ordering::Release);
            let diff =
```

```
                if after >= before {
                    after - before
                } else {
                    after + (u32::MAX - before)
                };
            diff
        }
    };
}
```

To get minimum, maximum and sum variables we used the same procedure as in SysTick.

```
/// KeyPair
let mut pk = [0u8; KYBER_PUBLICKEYBYTES];
let mut sk = [0u8; KYBER_SECRETKEYBYTES];
let bufs = Some(([1u8; 32].as_slice(), [255u8; 32].as_slice()));
let ticks = op_cyccnt_diff!(
    crypto_kem_keypair(&mut pk, &mut sk, &mut rng, bufs)
);
```

```
/// Encapsulate
let mut ct = [0u8; KYBER_CIPHERTEXTBYTES];
let mut ss = [0u8; KYBER_SSBYTES];

let encap_buf = Some([255u8; 32].as_slice());
let ticks = op_cyccnt_diff!(
    crypto_kem_enc(&mut ct, &mut ss, &pk, &mut rng, encap_buf)
);
```

```
/// Decapsulate
let mut ss = [0u8; KYBER_SSBYTES];
let ticks = op_cyccnt_diff!(crypto_kem_dec(&mut ss, &ct, &sk));
```

6.5 Code Coverage analysis

To ensure security and avoid weaknesses or bugs within the software, it is essential to have a measure that is able to evaluate the degree of both code coverage and test coverage of the application. One useful mechanism that measures and evaluates coverage in Rust is Source-based Code Coverage. To have correct reliability we need a test suite that can execute and verify the correctness) of every single line of code in the program.

6.5.1 Code Coverage

Code Coverage is a metric that evaluates the amount of lines of code effectively executed during the launch of a test suite and is very useful to understand how deeply the software code has been tested. So this process can be considered as a **white-box** testing approach, since it examines the the internal structures. In order to correctly execute a code coverage measurement, it is necessary to develop exhaustive test cases. This metric is expressed in percentage with the following formula:

$$\text{Code Coverage }\% = \frac{\text{Number of component's code lines effectively tested}}{\text{Total number of component's code lines}} \cdot 100$$

This metric is useful to identify specific areas on the program that are never reached by a set of test cases or eventually to delete needless test cases. In particular, it is useful to detect and resolve errors at early stages of a development cycle. Indeed, executing exhaustive tests to ensure confidence in the code base increases the software reliability. In the other hand, the code coverage metric is not an indication of flaws absence and code quality, it just measures the effectively execution of code and cannot measure the quality of tests launched.

6.5.2 Code Coverage Criteria

There are several levels of coverage differentiated according to the analysis criterion. The following list shows the main types.

- **Function coverage**: indicates how many functions of the component have been called at least once.
- **Statement coverage**: determine the number of lines of code within the component that have been executed.
- **Branch coverage**: inside a component flow can be present one of more paths (i.e if blocks introduces an alternative path based on the result of its condition).

This criteria determines whether all branch present in the component are exhaustively covered.

- **Condition coverag**e: also known as predicate or expression coverage. This coverage checks if each condition is evaluated true or false at least once.
- **Loop coverage**: loop is covered
  - if in at least one test the body was executed 0 times
  - if in some test the body was executed exactly once
  - if in some test the body was executed more than once

6.5.3 Test Coverage

Test Coverage is a software qualitative measure process that is used to understand if all possible scenarios have been considered, and so if all software features are covered by the developed test cases. Therefore, test coverage is a subjective evaluation of how many tests are able to correctly cover the potential software risks and the business specifications, i.e the three important IT documents: Functional Requirements Specifications (FRS), Software Requirements Specifications (SRS), User Requirements Specifications (URS). It uses a **black-box methodology** because it checks the application behavior from an external point of view, without considering internal logic. For this thesis purpose, we apply a test coverage to check the robustness of the library in case of a CCPA attack scenario. Since test coverage verifies the software operability, it can enhance the software quality. It also reduces the probability of introducing new weakness on the software. There are different types of test coverage differentiated according to the analysis criterion. The following list shows the various types.

- **Requirement coverage**: identifies whether all user requirements are correctly covered and met hence satisfy.
- **Product coverage**: check if the entire product works properly also in extreme conditions.
- **Risk coverage**: identifies potential software risks and ensures that adequate tests have been performed to mitigate these risks.
- **Boundary value coverage**: verifies the reaction of the component for boundary values as input (called test vectors).

In order to perform a correct test coverage, these elements must be executed:

- **Unit Testing**: tests to the smallest self-contained components of software that can be run in isolation and have no dependencies on any external factor (i.e queries to a database)
- **Functional Testing**: tests made on purpose to be able to guarantee the functionality of each point declared by the Functional Requirements Specification document.
- **Integration Testing**: in this phase each individual software modules (already tested using unit test) are combined and tested as a group.
- **Acceptance Testing**: this is the final phase of a testing procedure and is used to know whether a software product is ready to be released to public or not.

Chapter 7

Random Number Generator

Random numbers play a critical role in cryptography, particularly in generating session keys, initialization vectors (IVs), or cryptographic nonces. A cryptographic "number used once" is called "nonce", it is a random number that is used just once in a cryptographic communication. Kyber algorithms used nonces. A weak random number generator can compromise the security of any cryptographic protocol. In a software attack, adversaries attempt to exploit hidden periodicities or structures within a random sequence to guess the secret key and breach communication confidentiality. There are two primary classes of random number generators:

1. Deterministic RNG or Pseudo-RNG (PRNG): utilizes an algorithm to generate a sequence of bits from an initial value known as a seed. To ensure unpredictability, caution must be exercised in selecting seeds. The output of a PRNG is entirely predictable if both the seed and the generation algorithm are known. Since the generation algorithm is often publicly available, secrecy of the seed is crucial and it should be generated from a True RNG (TRNG).

2. Non-deterministic RNG or True RNG (TRNG): produces randomness based on an unpredictable physical source, known as the entropy source, which is beyond human control.

Therefore, there is a need for a standard certificate capable of certifying that hardware components like RNGs can be secure. For this purpose, the FIPS 140-2 standard has been introduced.

7.0.1 FIPS 140-2

Federal Information Processing Standard 140-2 (FIPS 140-2) [34] specifies the security requirements that will be satisfied by a cryptographic module.

7.1 The STM32F407VGT6 Board

The STM32F303VCT6 board adopted in this work does not provide RNG peripheral, so theoretical is necessary to assign the task of generating random numbers to an external hardware peripheral, that might be subject to interruptions or malfunctions.
For this reason, it was necessary to switch to the STM32F407VGT6 DISCOVERY model to conduct tests on a real integrated RNG component which provides a TRNG capability. The STM32F407VGT6 DISCOVERY board offers the following main hardware features:

1. Arm Cortex-M4 core
2. 1 MB Flash memory
3. 192 KB RAM
4. a TRNG peripheral

7.1.1 TRNG Functional Description

The board's RNG processor uses the continuous analog noise as a source for generating random numbers. It furnishes a random 32-bit value to the host when accessed. This component generates two consecutive random numbers every 40 periods of the RNG_CLK clock signal and monitors the RNG entropy in order to produce more stable sequence of values. The most important thing is that this board RNG component successfully completed the FIPS PUB 140-2 tests with a 99% success rate [36]. The NIST's FIPS PUB 140-2 verification is based either on the NIST statistical test suite:

1. NIST SP 800-22rev1a (April 2010): consists of 15 tests designed to assess the randomness of a binary sequence. These tests target different forms of non-randomness that may be present within the sequence.
2. NIST SP 800-90b (January 2018): to assess the quality of random generators intended for cryptographic applications using standardized methods to evaluate the reliability of an entropy source

7.1.2 TRNG Workflow

The RNG block diagram in figure 7.2 depicts from the bottom the analog circuit which comprises multiple ring oscillators whose outputs are combined using XOR operations to generate the seeds. The RNG_LFSR operates using a dedicated

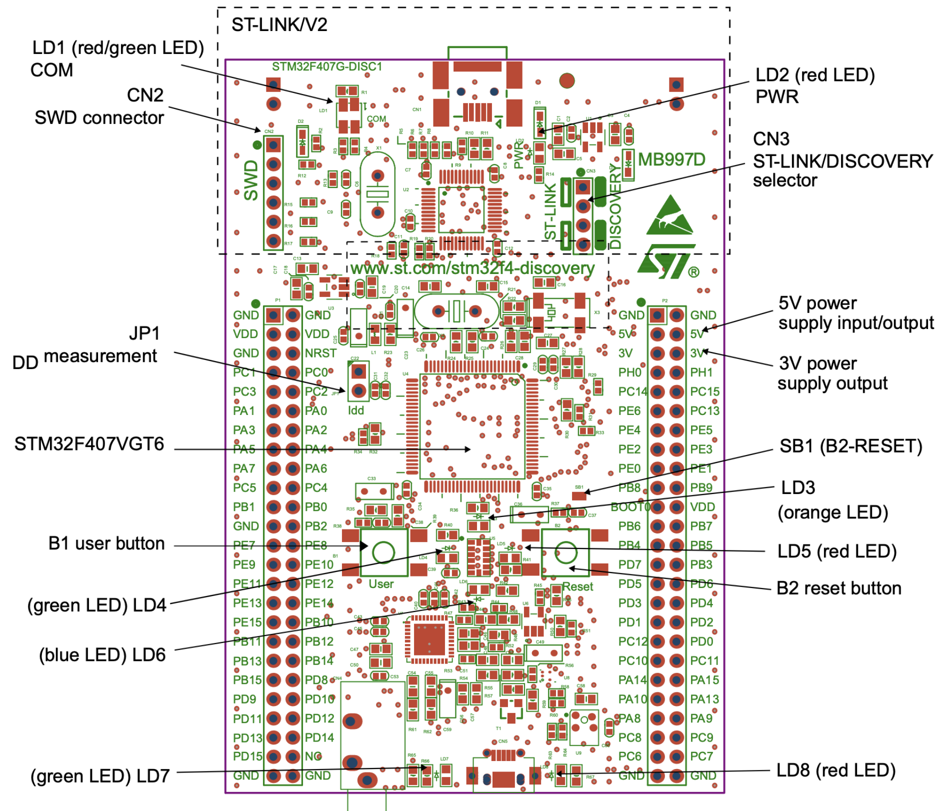


Figure 7.1: Front view of the STM32F407VGT6 board from [35]

clock (RNG_CLK) at a fixed frequency, ensuring that the randomness of the generated number remains consistent regardless of the system's primary clock frequency (HCLK). Once a sufficient number of seeds have been inputted into the RNG_LFSR, its contents are transferred to the data register (RNG_DR).

7.1.3 TRNG Error Management

There are two kind of errors that the module can detect and stored in a bit register:

- Clock error interrupt status (CEIS): If the bit is read as '1' it means that the RNG_CLK clock is not operating correctly
- Seed error interrupt status (SEIS): If the bit is read as '1' a detected fault indicates one of the following erroneous sequences:
  1. More than 64 consecutive bits of the same value (0 or 1)
  2. More than 32 consecutive alternations between 0 and 1 (0101010101...01)

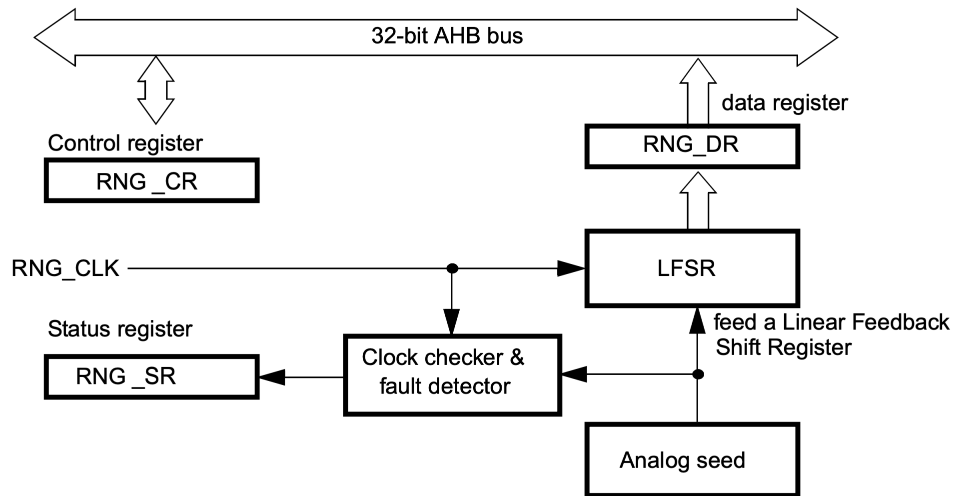


Figure 7.2: RNG Block Diagram from [36]

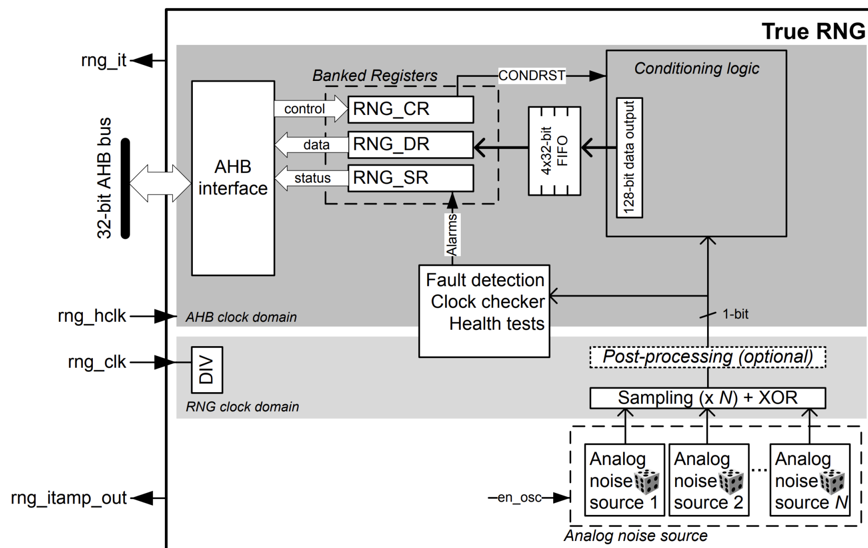


Figure 7.3: RNG Schema from [37]

Simultaneously, the analog seed and the dedicated RNG_CLK clock are monitored. Status bits within the RNG_SR register indicate any abnormalities detected in the seed sequence or if the RNG_CLK clock frequency falls below a specified threshold. In the event of an error, an interrupt can be triggered to notify the system.

7.2 Randomness Tests

We subjected the board to a test to verify if indeed each number has an equal probability of being selected. This is not an exhaustive test, the main purpose of this quick simple experiment is to verify and confirm what the manufacturer of the TRNG hardware component has already declared.

7.2.1 Test Description

The test consisted of:

- generating 40’000’000’000 random numbers
- each number must be within the range of 0 to 32000 (to avoid device stack overflow)
- each number will have a counter of how many times it was generated
- counters are represented on a vector of Rust’s unsigned 32 bit (u32) of 32000 elements (each number is 4 Bytes in size)
- the whole test was repeated 5 times

7.2.2 Results

The observed results were:

- the range of repetitions for each number spans from a minimum of 1’245’954 to a maximum of 1’254’804.
- all counts were represented on a bar graph in figure 7.4, where each value is declared in x-axis and the number of repetitions is placed on the y-axis
- it can be observed from the shape of the graph that they are distributed evenly, forming a straight line. There are no numbers with a significantly lower or higher probability of being selected compared to other values. We have realized that attempting to make the enlarged line more uniform in Figure 7.5 would require a significantly longer time, considering that this test is solely for the purpose of verifying and confirming what the manufacturer of the TRNG hardware component has already declared.
- no anomalies or instability of the module were detected during the test.
- it can be concluded that the module, as explicitly stated by the manufacturer, exhibits excellent randomness

svg

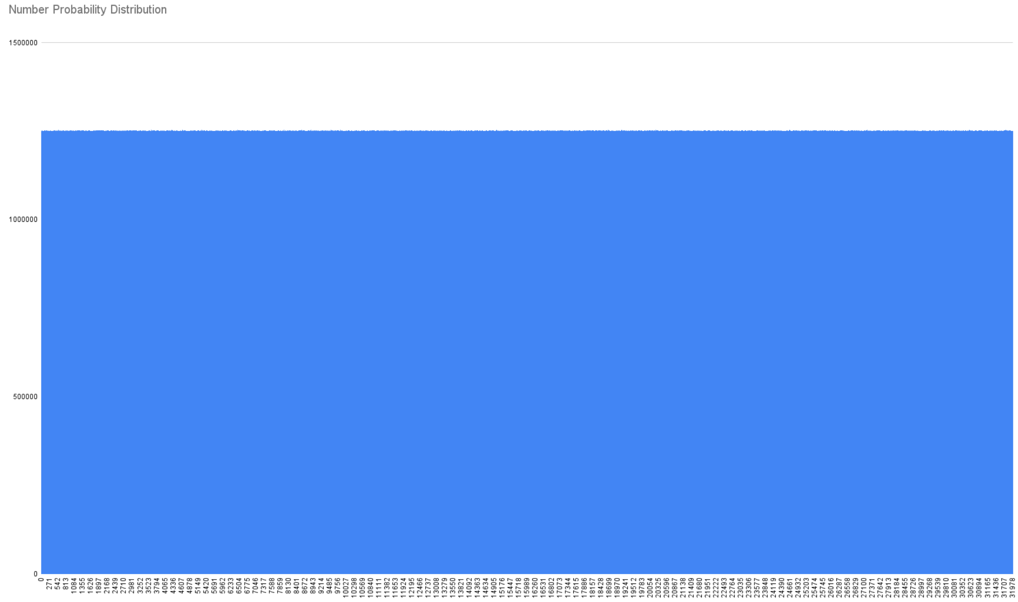


Figure 7.4: Bar chart depicting the probability distribution of each integer from 0 to 32000 considering 40 billion draws

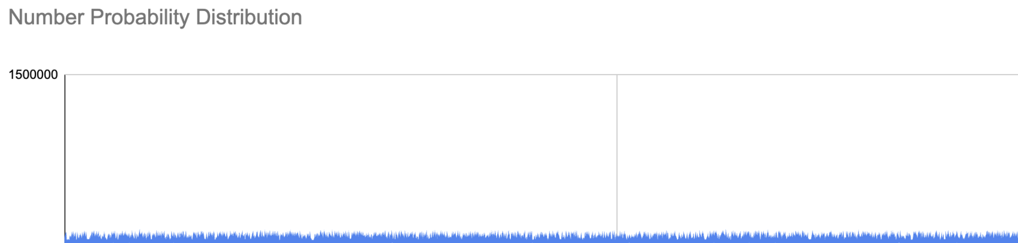


Figure 7.5: Enlarged surface of the previous graph

Chapter 8

Contributions to the library

8.1 Performance Tests for STM32F4 Board

Performance Tests were conducted following the approach used by pqm4 for its performance tests on Kyber written in C. We used the STM32F407VG board, which is the same board pqm4 used for their measurements as well.

It was observed that pqm4 utilized the open-source library LibOpenCM3, which provides a useful Hardware Abstraction Layer (HAL) for conducting these types of tests on ARM-Cortex M4 MCU.

In particular, we used the same configurations adopted by LibOpenCM3:

1. HSE (High-Speed External) clock = 8 MHz

2. PLL (Phase-Locked Loop) clock = 24 MHz

3. PLL48CLK = 48 MHz (required for the TRNG module)

Since Rust does not have a no_std library similar to LibOpenCM3, we utilized our custom SysTick and DWT methods explained in Chapter 6.4.2 for these measurements.

In Table 8.1, the results of the clock cycle measurements are presented for each KEM algorithm (KeyGen, Encapsulation, and Decapsulation).

In addition, we also performed the same experiment for the STM32F3 board, which does not have the TRNG peripheral. Therefore, we simulated a Pseudo Random Function for that purpose.

The results obtained have been represented on bar charts to make them easier to visualize and compare the numerical outputs.

Kyber Ver.	**Measure Method**	**Lang**	**Key Gen [cycles] (mean)**	**Key Gen [cycles] (min)**	**Key Gen [cycles] (max)**	**Encaps [cycles] (mean)**	**Encaps [cycles] (min)**	**Encaps [cycles] (max)**	**Decaps [cycles] (mean)**	**Decaps [cycles] (min)**	**Decaps [cycles] (max)**
512	pqm4	C clean	636181	635670	648917	843945	843433	856680	940320	939808	953055
768	pqm4	C clean	1059876	1057827	1071809	1352934	1350884	1364866	1471055	1469005	1482987
1024	pqm4	C clean	1649604	1646417	1686328	2016366	2013177	2053070	2159906	2156716	2196609
512	SysTick timer	Rust	782915	782912	782968	1075512	1075505	1075566	1037024	1037020	1037077
768	SysTick timer	Rust	1271579	1271574	1271630	1572956	1572949	1573007	1466918	1466912	1466965
1024	SysTick timer	Rust	1782451	1782444	1782497	2327229	2327220	2327277	2193613	2193604	2193659
512	DWT unit	Rust	784455	784455	784456	1077356	1077356	1077356	1042176	1042176	1042177
768	DWT unit	Rust	1278644	1278644	1278644	1571206	1571206	1571208	1468604	1468604	1468604
1024	DWT unit	Rust	1772551	1772551	1772551	2330897	2330896	2330898	2187373	2187373	2187374

Table 8.1: Speed comparison at 24MHz for the STM32F407VGT6 board based on the language and method used. The average, minimum and maximum value of each algorithms are reported for each KEM method

8.1.1 Results for STM32F4

In the graphs depicted in figure 8.1, 8.2 and 8.3, lower values indicates better performance. We noted that the C version is significantly faster for KeyGen and Encapsulation algorithms. For the Decapsulation function in the Kyber-768 and Kyber-1024 version, the Rust version is nearly as fast as the C version.

It is also important to emphasize that many performance improvements have been made to the C library, and the community has a larger group of developers compared to the Rust one. On the other hand, the Rust library is relatively young (currently at version 0.7.1) and requires further improvements. For being a new library written in Rust, it is very satisfying and promising according the results obtained. Considering all the advantages that Rust offers over C, it can be seen as a good trade-off between increasing security and maintaining a good performance. This is important, because often implementing stricter security measures implies the reduction in performance.

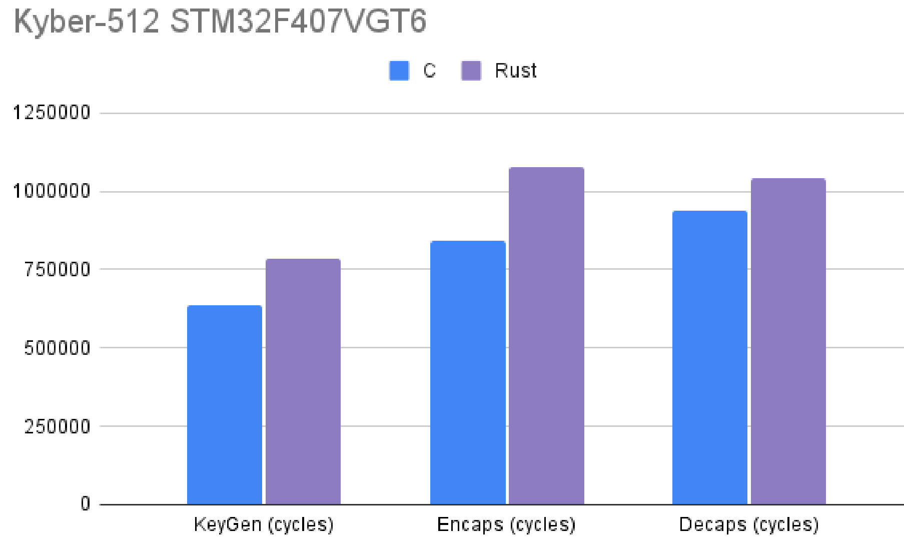


Figure 8.1: Kyber-512 KEM functions measured on STM32F407VGT6 board using DWT method

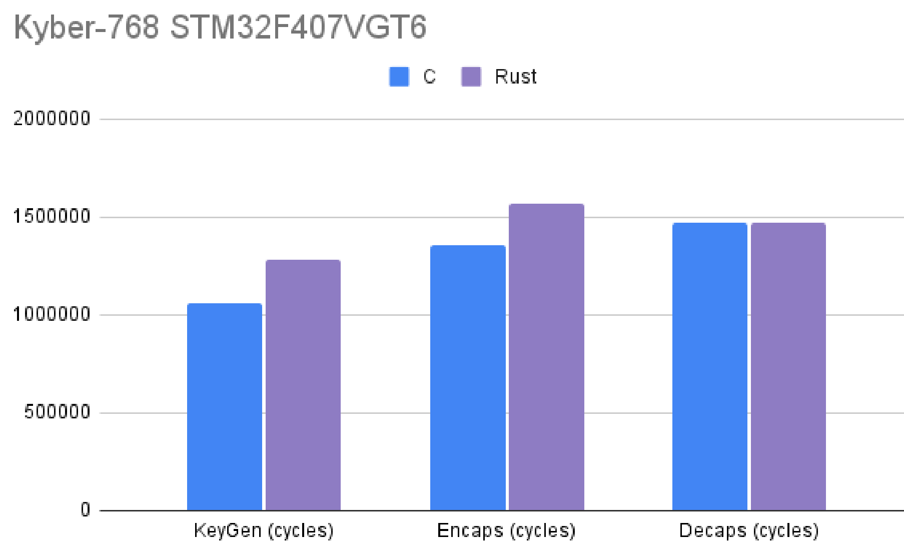


Figure 8.2: Kyber-768 KEM functions measured on STM32F407VGT6 board using DWT method

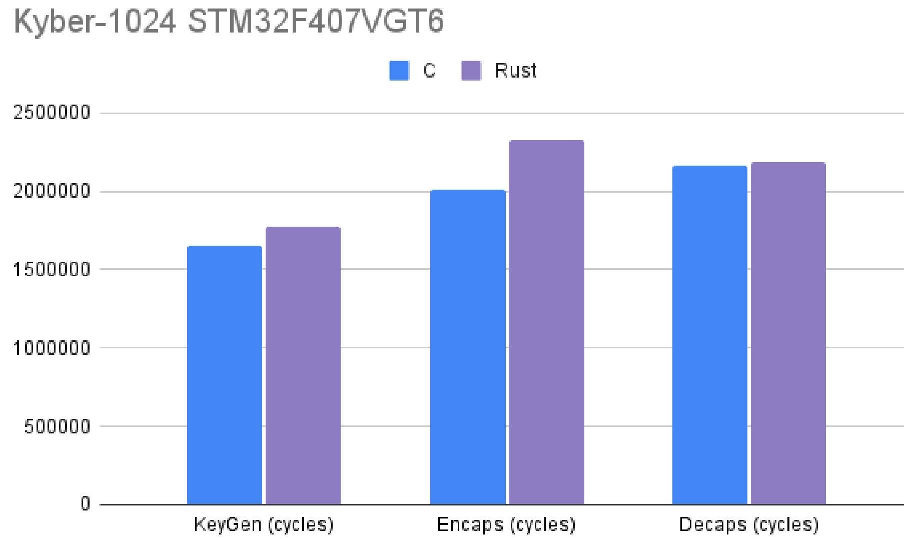


Figure 8.3: Kyber-1024 KEM functions measured on STM32F407VGT6 board using DWT method

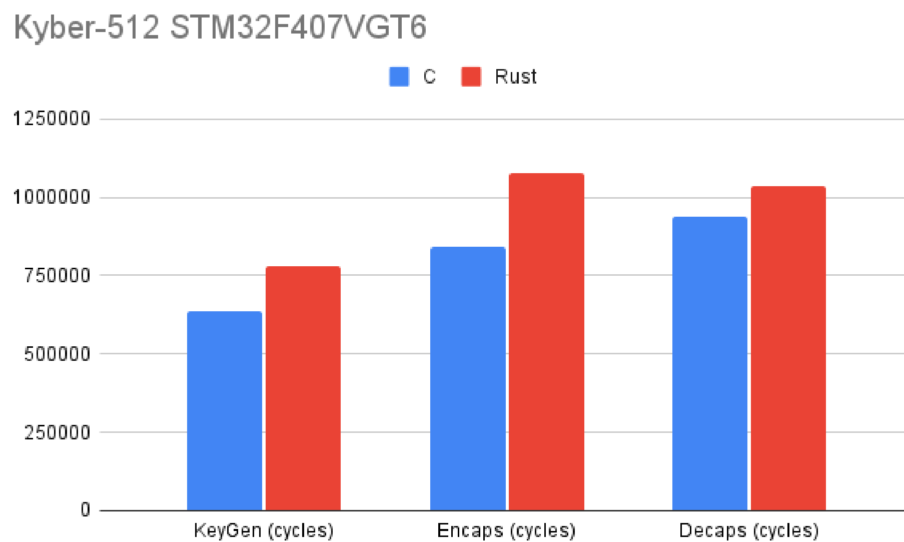


Figure 8.4: Kyber-512 KEM functions measured on STM32F407VGT6 board using SysTick method

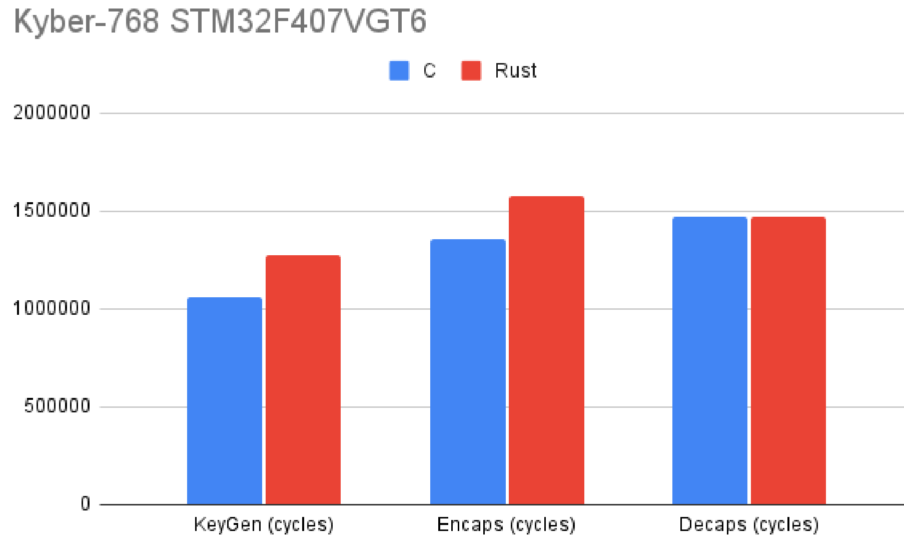


Figure 8.5: Kyber-768 KEM functions measured on STM32F407VGT6 board using SysTick method

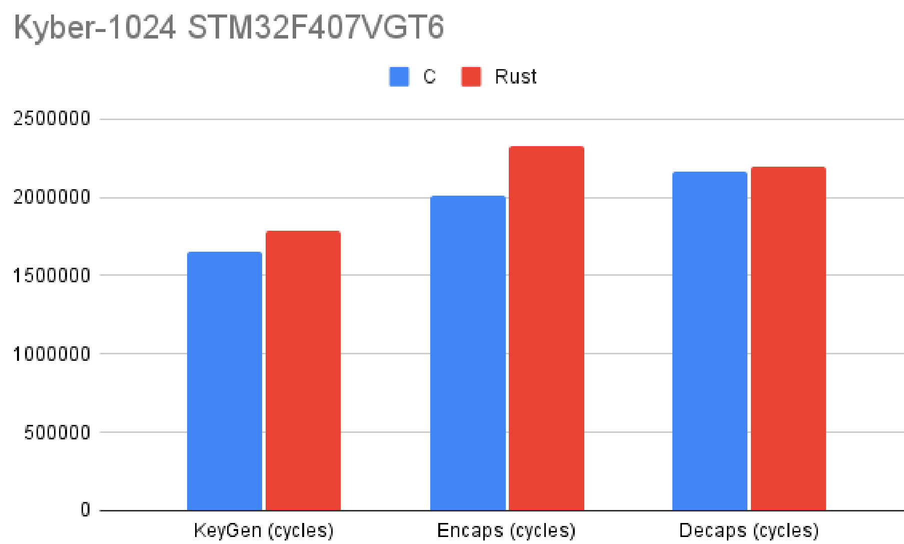


Figure 8.6: Kyber-1024 KEM functions measured on STM32F407VGT6 board using SysTick method

8.2 Performance Results for STM32F3 board

As already anticipated, the board examined for the analysis STM32F303VCT6 does not include the TRNG module. Hence, we used a custom pseudo-random function. The results were very similar to those obtained for the STM32F407VGT6 board presented in the previous paragraph.

Kyber Ver.	**Measure Method**	**Lang**	**Key Gen [cycles] (mean)**	**Key Gen [cycles] (min)**	**Key Gen [cycles] (max)**	**Encaps [cycles] (mean)**	**Encaps [cycles] (min)**	**Encaps [cycles] (max)**	**Decaps [cycles] (mean)**	**Decaps [cycles] (min)**	**Decaps [cycles] (max)**
512	pqm4	C clean	636181	635670	648917	843945	843433	856680	940320	939808	953055
768	pqm4	C clean	1059876	1057827	1071809	1352934	1350884	1364866	1471055	1469005	1482987
1024	pqm4	C clean	1649604	1646417	1686328	2016366	2013177	2053070	2159906	2156716	2196609
512	SysTick timer	Rust	676581	676579	676632	941811	941808	941860	921019	921016	921066
768	SysTick timer	Rust	1089819	1089815	1089866	1458933	1458927	1458979	1413732	1413727	1413780
1024	SysTick timer	Rust	1717910	1717903	1717956	2179686	2179677	2179729	2121991	2121983	2122037
512	DWT unit	Rust	676340	676340	676340	941908	941908	941908	921112	921112	921112
768	DWT unit	Rust	1091611	1091611	1091611	1461910	1461910	1461910	1413656	1413656	1413656
1024	DWT unit	Rust	1717095	1717095	1717096	2182045	2182045	2182045	2122381	2122381	2122381

Table 8.2: Speed comparison at 24MHz based on the language and method used. The average, minimum and maximum value of each algorithms are reported for each KEM method

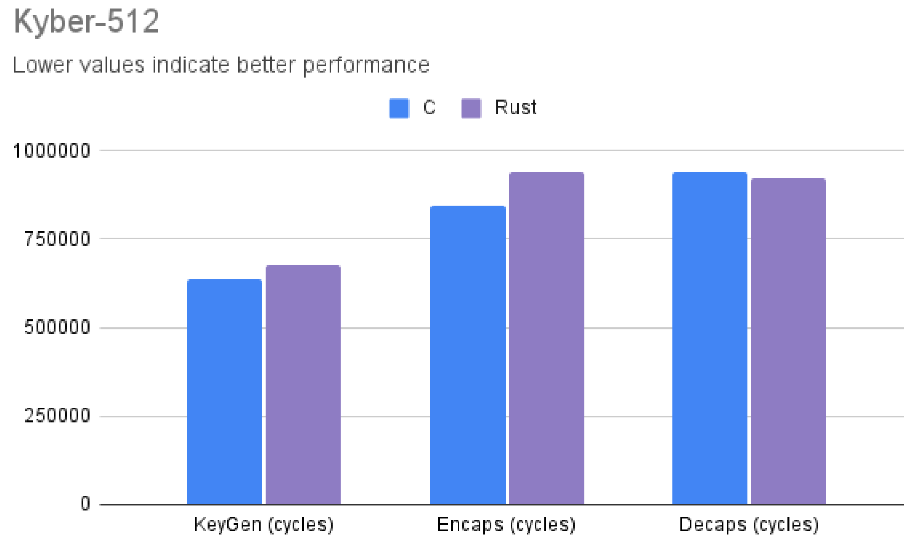


Figure 8.7: Kyber-512 KEM functions measured on STM32F303VCT6 board using SysTick approach

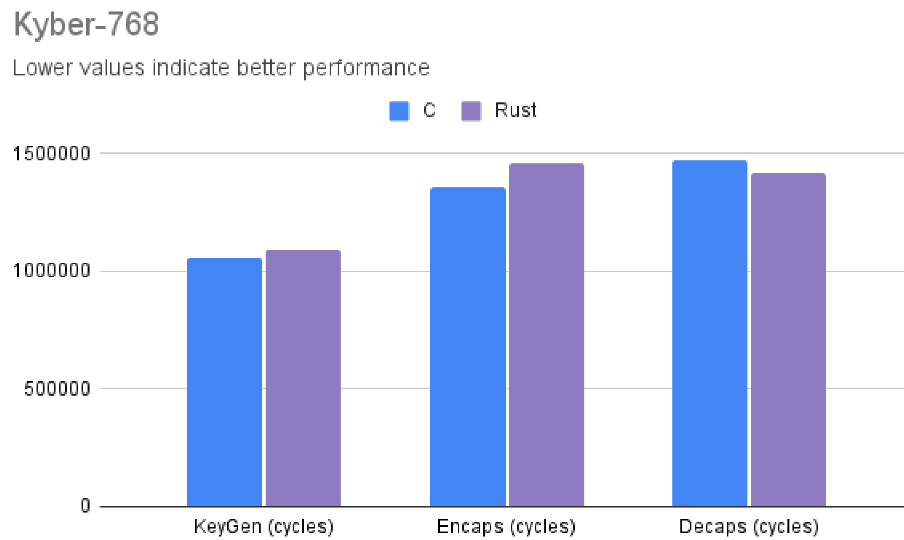


Figure 8.8: Kyber-768 KEM functions measured on STM32F303VCT6 board using SysTick approach

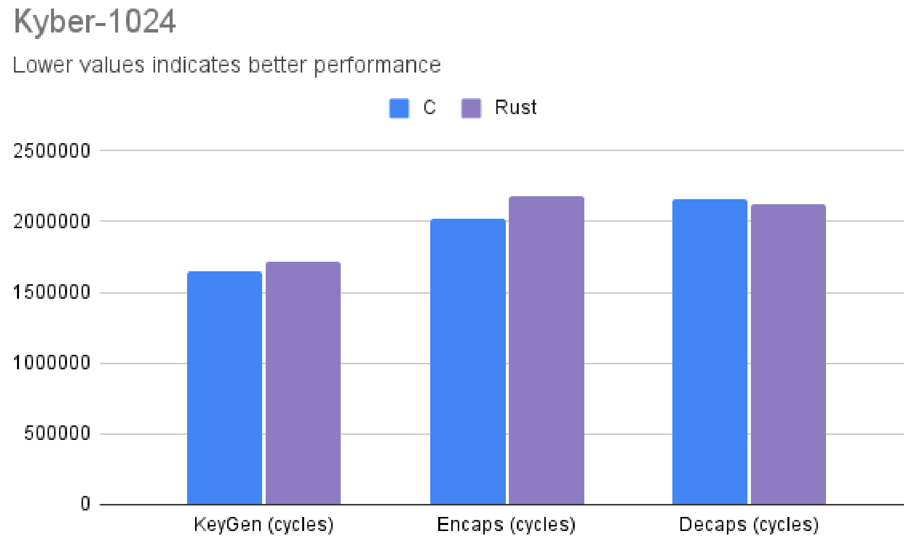


Figure 8.9: Kyber-1024 KEM functions measured on STM32F303VCT6 board using SysTick approach

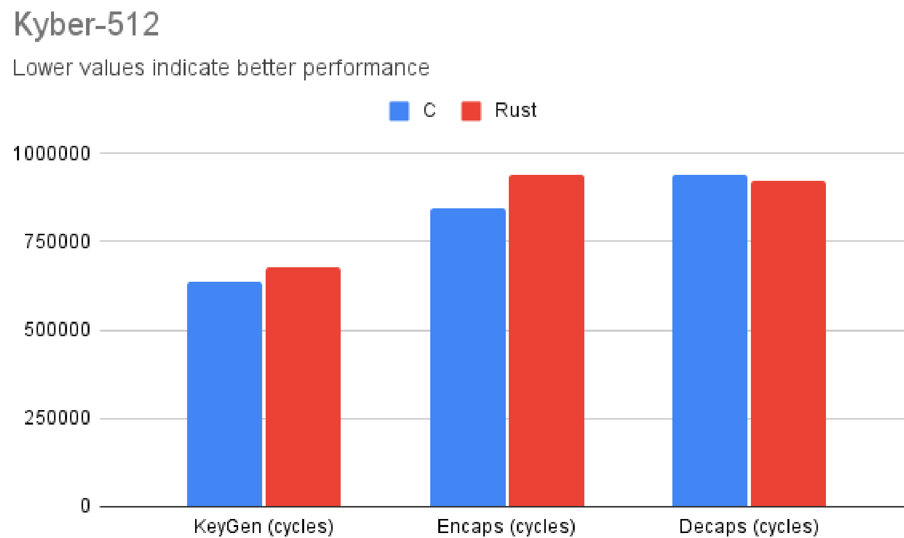


Figure 8.10: Kyber-512 KEM functions measured on STM32F303VCT6 board using DWT approach

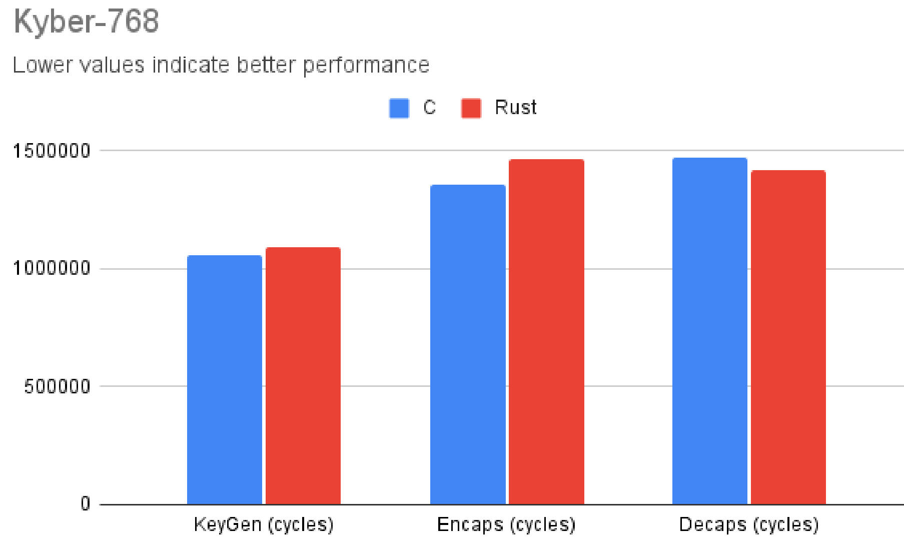


Figure 8.11: Kyber-768 KEM functions measured on STM32F303VCT6 board using SysTick approach

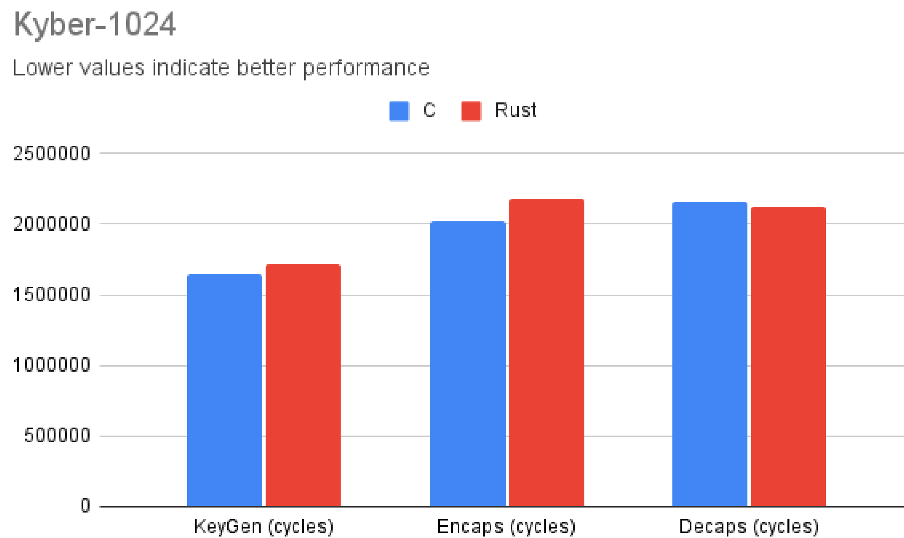
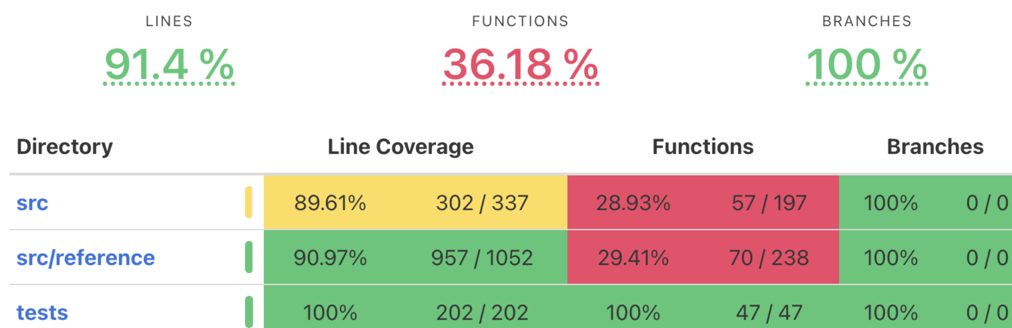


Figure 8.12: Kyber-1024 KEM functions measured on STM32F303VCT6 board using SysTick approach

8.3 Code Coverage Results

In order to measure the code coverage of the Kyber library we used the **grcov** tool, which is a Source-Code Coverage Tool developed by Mozilla. The results of the coverage analysis are shown in figures 8.13, 8.14 and 8.15. In general, the library gets a high coverage for most files.

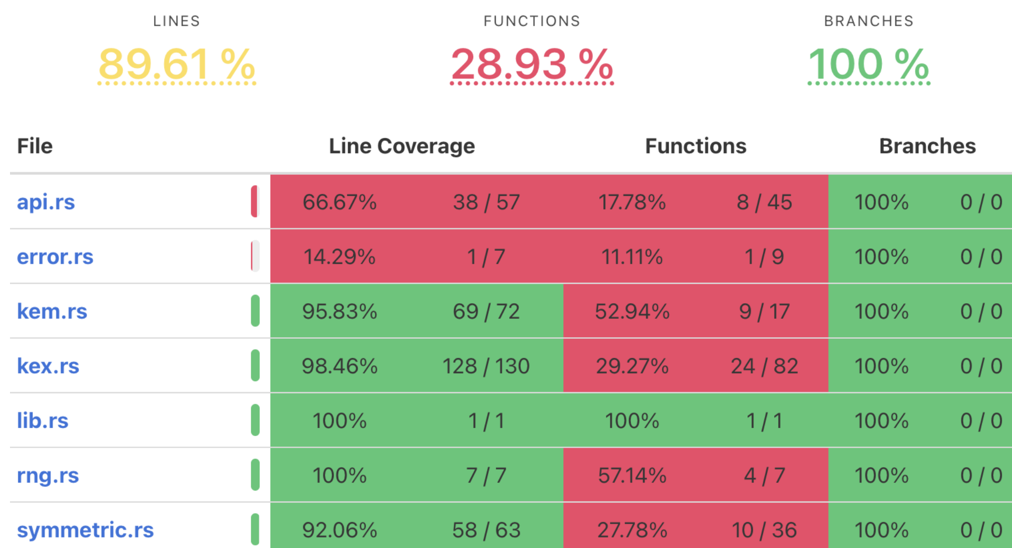
There are functions (in red) with low functional coverage, these cases are false negatives provided by grcov because functions are included in the final binary file depending on the selected Kyber version specified in the Cargo.toml file.



LINES	FUNCTIONS	BRANCHES
91.4 %	36.18 %	100 %

Directory	Line Coverage		Functions		Branches	
src	89.61%	302 / 337	28.93%	57 / 197	100%	0 / 0
src/reference	90.97%	957 / 1052	29.41%	70 / 238	100%	0 / 0
tests	100%	202 / 202	100%	47 / 47	100%	0 / 0

Figure 8.13: Coverage results for Kyber project root folder



LINES	FUNCTIONS	BRANCHES
89.61 %	28.93 %	100 %

File	Line Coverage		Functions		Branches	
api.rs	66.67%	38 / 57	17.78%	8 / 45	100%	0 / 0
error.rs	14.29%	1 / 7	11.11%	1 / 9	100%	0 / 0
kem.rs	95.83%	69 / 72	52.94%	9 / 17	100%	0 / 0
kex.rs	98.46%	128 / 130	29.27%	24 / 82	100%	0 / 0
lib.rs	100%	1 / 1	100%	1 / 1	100%	0 / 0
rng.rs	100%	7 / 7	57.14%	4 / 7	100%	0 / 0
symmetric.rs	92.06%	58 / 63	27.78%	10 / 36	100%	0 / 0

Figure 8.14: Coverage results for src folder

LINES	FUNCTIONS	BRANCHES
90.97 %	29.41 %	100 %

File	Line Coverage		Functions		Branches	
cbd.rs	58%	29 / 50	22.22%	4 / 18	100%	0 / 0
fips202.rs	90.8%	375 / 413	22.73%	15 / 66	100%	0 / 0
indcpa.rs	97.81%	179 / 183	37.21%	16 / 43	100%	0 / 0
ntt.rs	100%	56 / 56	33.33%	4 / 12	100%	0 / 0
poly.rs	82.18%	143 / 174	31.48%	17 / 54	100%	0 / 0
polyvec.rs	99.3%	142 / 143	30.3%	10 / 33	100%	0 / 0
reduce.rs	100%	17 / 17	33.33%	2 / 6	100%	0 / 0
verify.rs	100%	16 / 16	33.33%	2 / 6	100%	0 / 0

Figure 8.15: Coverage results for src/reference folder

8.4 Contributions to the library

The following two paragraphs described our contributions to the library during our study. All these contributions have been officially released in the library.

Procedure Our changes have been reviewed by the library authors which basically use pipelines in order to execute automated tests (i.e., for verifying the compatibility on the main environments such as linux, Mac OS, Windows, etc.). After this check, our pull request can be finally merged to the master branch.

8.4.1 Solution for the Unhandled RNG exceptions

The library function *randombytes()* is responsible for filling a vector with bytes of random numbers. This function, internally, calls the *fill_bytes()* method (from the RngCore trait).

Our solution consists of using method **try_fill_bytes()** (coming from RngCore trait as the previous) because this function is declared to be safer than the previous one because it is capable of reporting the type of error without terminating the process. Therefore, function **try_fill_bytes()** ensures a proper error handling (returns *Resut<(), Error>* on failure) so it is specially designed for devices which RNG peripheral can fail.
Lastly, to be consistent and follow the error pattern adopted by the library, we

added a new type that we called *RandomBytesGenerator* (as in figure 8.16 and 8.17) to the set of *KyberErrors*. We also made sure that in case this exception occurs, the error is propagated up to the caller function as *Decapsulation* and *InvalidInput* errors were also handled within the library. The pull request of this proposal was accepted and released.

```
3 src/error.rs
@@ -7,13 +7,16 @@ pub enum KyberError {
7   7      /// The ciphertext was unable to be authenticated.
8   8      /// The shared secret was not decapsulated.
9   9      Decapsulation,
    10 +   /// Error trying to fill random bytes (i.e external (hardware) RNG modules can fail).
    11 +   RandomBytesGeneration,
10  12   }
11  13
12  14   impl core::fmt::Display for KyberError {
13  15     fn fmt(&self, f: &mut core::fmt::Formatter) -> core::fmt::Result {
14  16       match *self {
15  17         KyberError::InvalidInput => write!(f, "Function input is of incorrect length"),
16  18         KyberError::Decapsulation => write!(f, "Decapsulation Failure, unable to obtain shared secret from ciphertext"),
    19 +       KyberError::RandomBytesGeneration => write!(f, "Random bytes generation function failed"),
17  20       }
18  21     }
19  22   }
```

Figure 8.16: Added a new error enum called RandomBytesGenerator

```
8 src/rng.rs
@@ -1,9 +1,13 @@
1   1    use rand_core::*;
    2  + use crate::KyberError;
2   3
3   4    // Fills buffer x with len bytes, RNG must satisfy the
4   5    // RngCore trait and CryptoRng marker trait requirements
5      - pub fn randombytes<R>(x: &mut [u8], len: usize, rng: &mut R)
    6  + pub fn randombytes<R>(x: &mut [u8], len: usize, rng: &mut R) -> Result<(), KyberError>
6   7      where R: RngCore + CryptoRng,
7   8    {
8      -   rng.fill_bytes(&mut x[..len]);
    9  +   match rng.try_fill_bytes(&mut x[..len]) {
    10 +     Ok(_) => Ok(()),
    11 +     Err(_) => Err(KyberError::RandomBytesGeneration)
    12 +   }
9   13   }
```

Figure 8.17: Modified the *randombytes()* function in order to use the try_fill_bytes() method and handle the error case

8.4.2 Solution advantages

Figure 7.12 depicts the IND-CCA scheme for the key generation. This scenario represents the worst-case scenario because the failure of the TRNG module occurs after the execution of the most computationally intensive functions (IND_CPA_KeyPair and SHA3-256).

More in detail, the processed data is progressively stored in the stack as it is generated and processed. At this point, when the second call for getting random numbers (TRNG) fails, it throws a panic event which is responsible for freeing the stack and abort the process. This scenario represents a non-negligible issue, because IoT devices are specially designed to be used intensively, leading to a higher probability of failure with service disruption.

This solution depicted in figure 8.19, brings these advantages to the library:

1. Security: TRNG peripheral (as declared on the reference manual [36]) can generate unstable outputs that can be exploited for predictive attacks.
2. Failure Recovery: it allows the library to recover from failures without losing the already processed and stored data, by giving a second chance to the TRNG or trying an alternative software path or fallback mechanisms
3. Robustness: it guarantees that unexpected errors do not abort the whole process
4. Error Reporting and Debugging: meaningful and readable error messages or logging information can be generated, simplifying the debugging process and accelerating bug resolution

The same type of solution can be applied throughout the library wherever there is a need for random numbers generation, such as Encapsulation: issue as in figure 8.20 and solution shown in figure 8.21.

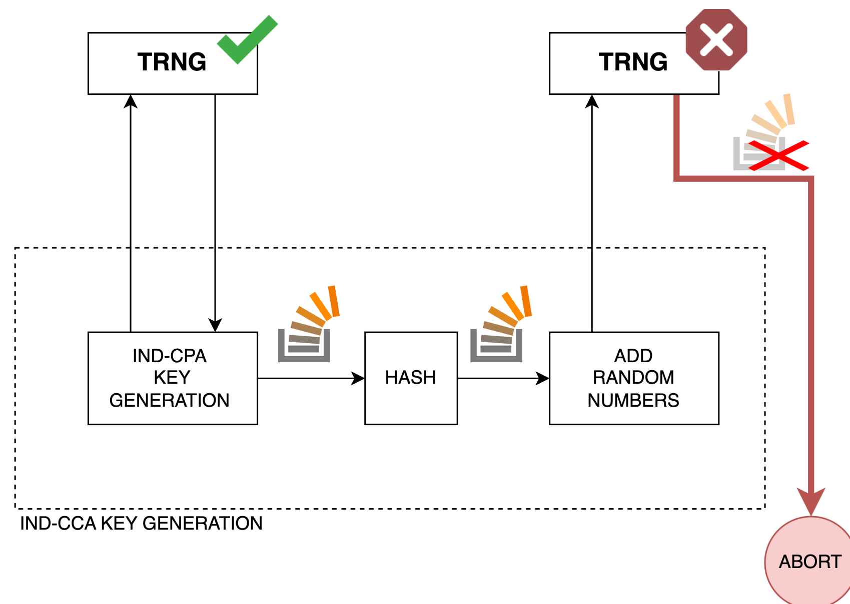


Figure 8.18: IND-CCA Key Generation RNG issue

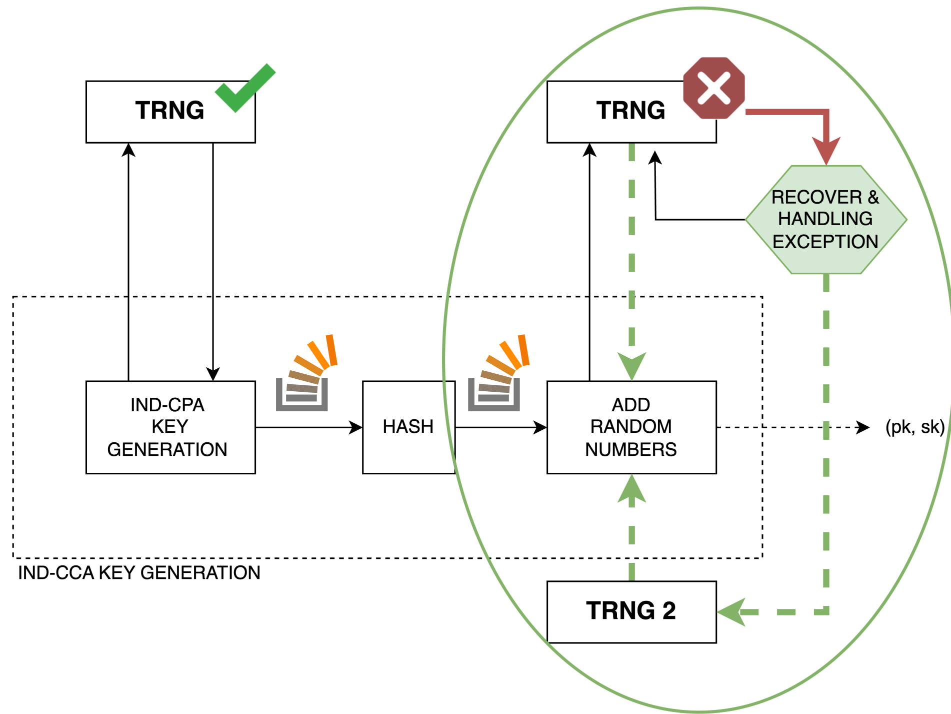


Figure 8.19: Our IND-CCA Key Generation RNG solution

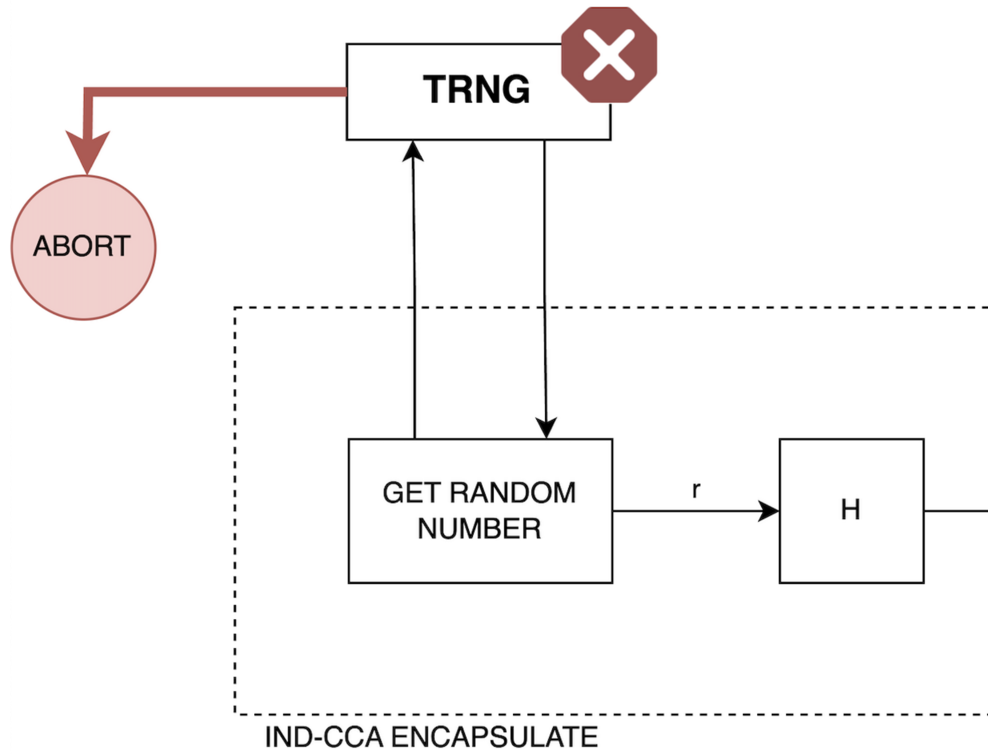


Figure 8.20: IND-CCA Encapsulation RNG issue

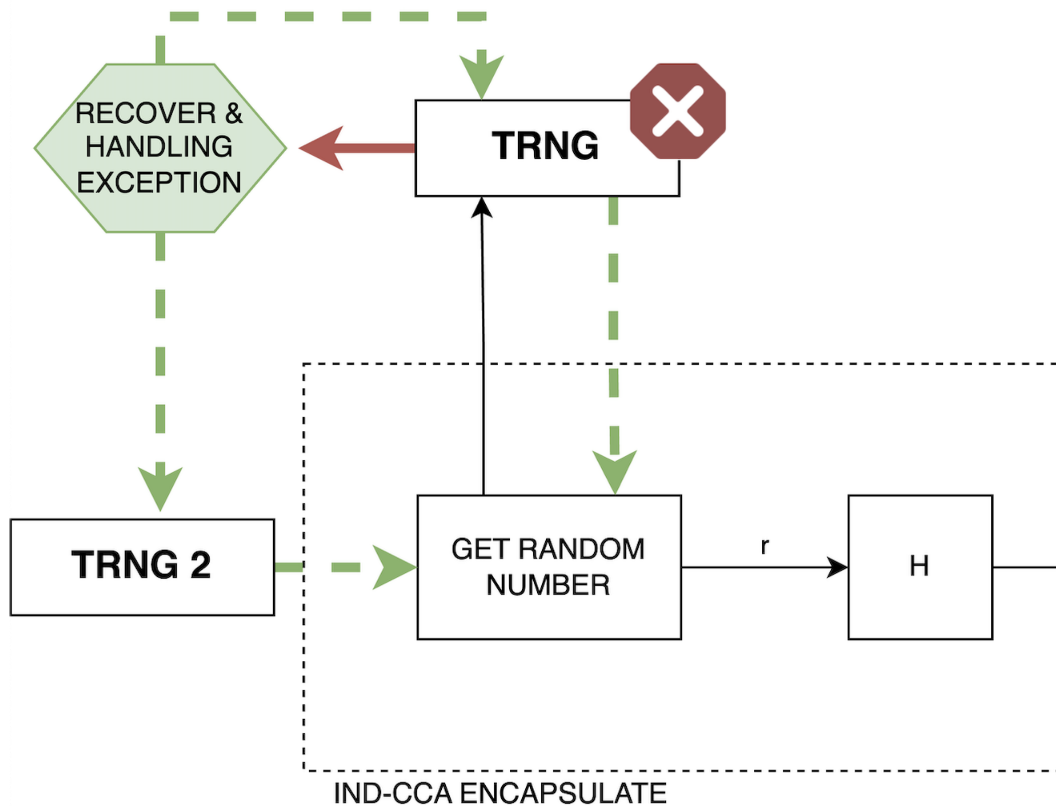


Figure 8.21: Our IND-CCA Encapsulation RNG solution

8.4.3 Contributions to enhance Code Quality

We notice that the *store64()* function was created to store a 64-bit integer as a byte array in little-endian order. In order to have a **single point of responsibility and failure** for a certain type of functionality, an enhanced code **reuse** and a more **readable** algorithm we proposed to use *store64()* instead of using a custom logic as in figure 8.22.

```
8 src/reference/fips202.rs
@@ -499,10 +499,12 @@ pub(crate) fn keccak_squeeze(
499 499           pos = 0
500 500         }
501 501         let mut i = pos;
    502 +       let mut w = i/8;
502 503         while i < r  && i < pos+outlen {
503     -         out[idx] = (s[i/8] >> 8*(i%8)) as u8;
504     -         i += 1;
505     -         idx += 1;
    504 +         store64(&mut out[idx..], s[w]);
    505 +         i += 8;
    506 +         w += 1;
    507 +         idx += 8;
506 508         }
507 509         outlen -= i-pos;
508 510         pos = i;
```

Figure 8.22: Merged pull request `https://github.com/Argyle-Software/kyber/commit/5e931d6d8ca536a98308c2faa0b505f3e2c9949b`

Branch Coverage

During the code coverage analysis process, some areas were not covered by the library's test suite available before March 16, 2023. In particular, the most important was the absence of tests that covered a branch concerning the validation of the public key vector dimension, which is input to the KEM's encapsulate function, and the private key, which is input to the decapsulate function. Our proposed solution consisted in very basic tests as shown in figure 8.23, but of significant importance. Another minor contribution was the correction of the library's README as in figure 8.24.

```
18 tests/kem.rs
@@ -18,5 +18,23 @@ fn keypair_encap_decap_invalid_ciphertext() {
18 18      assert!(decapsulate(&ct, &keys.secret).is_err());
19 19    }
20 20
   21 + #[test]
   22 + fn keypair_encap_pk_wrong_size() {
   23 +   let mut rng = rand::thread_rng();
   24 +   let pk: [u8; KYBER_PUBLICKEYBYTES + 3] = [1u8; KYBER_PUBLICKEYBYTES + 3];
   25 +   assert!(encapsulate(&pk, &mut rng).is_err());
   26 + }
21 27
   28 + #[test]
   29 + fn keypair_decap_ct_wrong_size() {
   30 +   let ct: [u8; KYBER_CIPHERTEXTBYTES + 3] = [1u8; KYBER_CIPHERTEXTBYTES + 3];
   31 +   let sk: [u8; KYBER_SECRETKEYBYTES] = [1u8; KYBER_SECRETKEYBYTES];
   32 +   assert!(decapsulate(&ct, &sk).is_err());
   33 + }
22 34
   35 + #[test]
   36 + fn keypair_decap_sk_wrong_size() {
   37 +   let ct: [u8; KYBER_CIPHERTEXTBYTES] = [1u8; KYBER_CIPHERTEXTBYTES];
   38 +   let sk: [u8; KYBER_SECRETKEYBYTES + 3] = [1u8; KYBER_SECRETKEYBYTES + 3];
   39 +   assert!(decapsulate(&ct, &sk).is_err());
   40 + }
```

Figure 8.23: Merged pull request https://github.com/Argyle-Software/kyber/commit/5e931d6d8ca536a98308c2faa0b505f3e2c9949b

```
2 tests/readme.md
@@ -11,7 +11,7 @@ Which will clone the C reference repo, generate the KAT files, then rename and p
11 11
12 12    To run the known answer tests you will need to enable `kyber_kat` in `RUSTFLAGS`. To check different Kyber le
           to include those flags also. eg:
13 13    ```bash
14    - RUSTFLAGS=' --cfg kyber-kat' cargo test --features "kyber1024 90s"
   14 + RUSTFLAGS=' --cfg kyber_kat' cargo test --features "kyber1024 90s"
15 15    ```
16 16
17 17    For applicible x86 architectures you must export the avx2 RUSTFLAGS if you don't want to test on the referenc
```

Figure 8.24: Merged pull request https://github.com/Argyle-Software/kyber/commit/5e931d6d8ca536a98308c2faa0b505f3e2c9949b

Boundary Value Coverage Observations

The library's *ntt()* is not considered an autonomous function because its correct processing depends on the input values passed as a parameter vector, therefore it depends on external factors. In particular, although it accepts as input a mutable vector of **i16** (that means values from -32768 to 32767), we must take into account that in order to correctly execute the *ntt()* in a Kyber context we must only consider a vector of elements of $\mathbb{Z}_q$:

$$0 \leq r < q - 1$$

or equivalently

$$-q/2 \leq r < q/2$$

where q is equal to 3329.
Hence, not all integers in the i16 type range can be considered valid input values.
We tested all possible input values, an **overflow** occurs when trying to add at line 17 when we test the *ntt()* with a sample test vector (reported in Appendix A), populated with random values. In particular, in our example the range is $-31300 < r < 7213$ for which elements are still contained within the i16 type range. Same reaction can be obtained with other set of random elements.
There is no range validity check inside the function, because it is assumed that the input values respects that theoretical range ($-\frac{q}{2} \leq r \leq \frac{q}{2}$). The same assumption is adopted for the *montgomery_reduce()*, *poly_compress()* and *barrett_reduce()* function: they are declared to accept integers with a certain range (i.e i32 for the first function and i16 for the other functions), but not all values are the effective inputs. Consequently, when tested in a Kyber context they don't give any kind of errors, but when run without the theoretical Kyber constraints they trigger overflow errors.
Although it's not absolutely necessary, it could be interesting and useful to define an input range validity check, in order to prevent overflow cases and thus make the function safe and completely independent of the external parameters.

All tests made are reported in Appendix A.

```
pub fn ntt(r: &mut[i16])
{
  let mut j;
  let mut k = 1usize;
  let mut len = 128;
  let (mut t, mut zeta);

```

```
8    while len >= 2 {
9      let mut start = 0;
10     while start < 256 {
11       zeta = ZETAS[k];
12       k += 1;
13       j = start;
14       while j < (start + len) {
15         t = fqmul(zeta, r[j + len]);
16         r[j + len] = r[j] - t;
17         r[j] += t;
18         j += 1;
19       }
20       start = j + len;
21     }
22     len >>= 1;
23   }
24 }
```

Chapter 9

Conclusions

To thoroughly understand how the CRYSTALS-KYBER library works, it was crucial to dedicate a careful study to the mathematical aspects of the Post Quantum Cryptography. The experiments conducted with the boards required meticulous documentation of the used hardware components, an analysis of the Rust language and its effects when adopted within embedded systems.

In conclusion, our way of working allowed us to realize solutions that improve the software quality and maturity of the library. Among our contributions, probably the most significant one was related to enhance prevention and management of errors or malfunctions of the RNG peripheral.

Every single solution has been accepted by the library community.

An effort was made to provide tests regarding the impacts of adopting the Rust language for this PQC library (compared to the tests conducted in the C version), which are particularly new to the current state of the art.

Technology continues to evolve, and two significant developments are currently unfolding and warrant attention as possible future work:

- The Rust community is continuously growing, as is the adaptation of the language for embedded systems. This strengthens support from more developers, so faster language improvements

- In August 2023, the NIST presented the first draft of FIPS 203 concerning the Module-Lattice-based Key-Encapsulation (ML-KEM). This basically corresponds to the Kyber library studied in this thesis but with modifications to make it more secure.

These elements are crucial for the continuation of research regarding aspects of our work.

Appendix A

Boundary Value Coverage Observations

test_vector_0 = [-21371, 843, -19833, -2213, -20636, 4004, -23020, 1026, -18979, 3760, -20413, 3228, -18618, 2781, -18186, 403, -20153, -1048, -17895, 2208, -17776, -856, -17064, 1116, -19321, -953, -20035, -2851, -21259, -3473, -21697, -743, -20172, 1835, -23106, 3161, -25347, -1455, -22727, 587, -20590, 529, -20898, -1577, -21529, 1165, -19679, 3219, -21803, -1935, -19169, -4397, -17704, -874, -17216, 438, -21326, -3238, -19556, -236, -19556, 1087, -20662, 443, -18150, 784, -18154, -794, -20483, -814, -20813, -2668, -22937, -1092, -21109, -4168, -19738, -2044, -19512, -500, -18081, 2730, -14921, -500, -18412, 3547, -19338, 4411, -21684, 1430, -19832, 850, -18139, -843, -17673, 1431, -20373, 643, -22793, -1957, -20567, -308, -21219, 2210, -22084, -2782, -20250, -1718, -19992, 533, -18250, -2813, -22432, -381, -23164, -983, -22799, -2753, -22105, 317, -23205, -414, -23537, 1872, -24616, 2191, -26038, 3911, -19965, -485, -19639, -3721, -21280, -4350, -22828, -3236, -20392, 2087, -20610, -817, -21168, -1161, -18454, -1837, -23320, 5611, -24960, 2491, -20434, -83, -21674, 1761, -20192, -863, -23372, 591, -17922, -761, -19230, 1157, -23144, 2402, -22486, 2, -20100, 610, -21166, 1354, -21660, 974, -18734, 562, -22566, 949, -23632, -589, -22349, 2897, -23745, 2711, -22132, -452, -19122, -80, -20258, 7213, -22534, 5119, -22240, 1592, -24452, 4720, -23673, 69, -24617, 211, -28033, -1977, -26597, -2071, -27027, -4698, -26273, -1414, -23736, -5232, -27020, -4152, -25794, 419, -26306, -939, -28749, -2933, -26207, -4907, -27645, 2055, -29259, 3957, -31300, -410, -29683, 3255, -26245, -1236, -5374, 4176, -22983, -4304, -2648, 1122, -21719, -654, -4911, 5995, -23909, -4114, -7847, 2435, -26645, -1817, 1713, -2366, -23591, -3543, -807, 540, -23486, -6846, -1553, 2366, -23198, -5598, -3293, 3140]

N° Executions with random coeffs for each iteration	Coeffs Range	r[j] after fqmul				r[j + len] after fqmul				input Montgomery_reduce				output Montgomery_reduce			
		Max	Min	Input for Max	Input for Min	Max	Min	Input for Max	Input for Min	Max	Min	Input for Max	Input for Min	Max	Min	Input for Max	Input for Min
1000	[-3..3]	9306	-9953	[-1, -1, -3, 3, -2, 1, -1, 3, -1, 1,	[-3, 2, 3, 1, -1, -1, -1, 1, 1, -1,	10377	-9586	[-2, 1, 2, 3, -3, 3, 1, -3, 3, 1, -2	[0, -1, 2, -3, 1, -2, 0, 2, 3, 2, 0,	14402600	-13230525	[3, 3, 2, 0, 2, -1, -3, 2, 0, -1, 0,	[2, -3, 0, 0, 2, 0, -1, 3, -3, -2, -3	-1698 size 3566)	-1802	[3, 3, -3, 0, -2, -2, 3, 0, -1, 1, -1	[3, 2, 0, 3, 2, -2, 0, 3, -2, 3, -1,
10000	[-3..3]	10766	-10962	[3, -2, 1, 3, -1, 1, -3, 1, -2, 1, 0	[0, -3, 1, -2, -3, 3, 0, -3, 2, -1,	10551	-10441	[1, 2, -1, 1, 1, 2, 2, -1, -1, -3, 3	[-3, 2, -2, -2, 2, 2, -3, -1, -3, 3,	14331284	-14424078	[2, 1, 1, 0, -3, -1, -2, -2, -1, -2,	[-1, 0, -2, -1, 0, 3, -1, -3, -1, -3,	-1717 size 3549)	-1856	[2, 1, -3, -2, -2, 1, -3, 1, -1, -1,	[-2, 3, -2, -1, 2, 2, 1, 1, 2, -2, 3
100000	[-3..3]	10653	-10649	[-3, 1, -2, -1, -1, -1, 0, -3, 3, 0,	[-2, 2, 1, 0, -2, 0, 3, 3, 0, 2, 2,	10753	-10724	[-3, 2, 0, -3, 0, 1, -2, -1, -2, 2,	[3, -1, 0, 3, 2, 2, 0, -1, -1, -1, 3	15521670	-14997360	[0, -3, 1, 2, 1, 3, 0, -3, -2, -3, 3,	[1, -1, 2, 3, 1, -1, -1, 0, -2, -1, -	-1771 size 3633)	-1862	[3, 1, -3, 1, -3, 1, -1, 1, -1, -3, 1	[3, -2, 1, -3, 2, 1, 2, 0, -2, 2, 1,
1000000	[-3..3]	11200	-11076	[1, -2, 0, 0, 3, 0, -1, 3, -3, -2, 3	[-2, -2, 0, 0, 2, -2, -1, 0, 2, 2, -	11079	-11041	[2, -2, 2, -2, -2, 2, -3, -2, -3, -2	[3, 2, 3, -1, -2, -1, 2, 1, -1, -2,	15901515	-15394389	[3, 2, 0, -3, 0, 2, 3, -2, 1, -1, 2,	[-3, 2, -2, -1, 3, -2, 0, -2, -3, -1,	1875	-1882	[0, -3, 1, -2, 0, 2, -3, -3, 0, -1, 0	[2, -1, 3, -3, -3, 2, -3, -1, 1, 0,
10000000	[-3..3]	11167	-11205	[2, -3, -1, 2, 2, 3, 1, -1, 2, 0, 3,	[1, 2, 1, 0, -1, -1, 2, -1, -1, -3, 3	11384	-11281	[-2, 1, 3, 3, 0, 0, -3, 2, 0, 3, -2,	[-2, 1, 3, 1, -3, -3, 3, -3, 3, 2, -	15877065	-16248246	[3, 3, 2, 1, 3, -3, -1, 0, -2, -1, 3,	[-1, -1, 2, -1, -1, -3, 2, -1, 0, -2,	1882	-1881	[0, -2, 3, 2, 2, -1, 3, 3, -2, 1, 1,	[2, 3, 3, 3, 2, -3, -1, 3, 0, 2, 0,
100000000	[-3..3]	11483	-11374	[2, 0, -3, -1, -2, 1, 0, -1, 1, -1, 0	[-2, 3, -3, 3, 1, 3, -3, 0, -1, -1, 2	11332	-11375	[-3, 3, 1, -2, 3, 0, 0, -1, -1, 1, -2	[3, 2, -1, -3, 0, -1, 2, -3, 2, -1, -	16093959	-16228338	[-2, 0, 3, -3, 3, 2, -2, -1, 1, -2, 2	[1, 3, 0, -2, -2, -2, 2, -2, -3, -2, -	1888	-1890	[-2, 0, 3, -3, 3, 2, -2, -1, 1, -2, 2	[1, 3, 0, -2, -2, -2, 2, -2, -3, -2,
		11483	-11374			11384	-11375			16093959	-16248246			1888	-1890		
10000	[-(q-1)/2..(q-1)/2]	11196	-10710	[1442, 80, 708, -729, -314, -44	[740, -1385, -7, 524, 1204, -33	11316	-11217	[-251, 1272, -1096, 1300, 134	[-1544, 1346, -1380, 1285, 12	16224360	-15799740	[-1243, -927, -1481, -1499, -46	[-590, -844, -369, -1513, 542,	1840	-1859	[-52, 1500, -1049, 1235, 1564,	[822, 948, -761, 424, -318, 34
100000	[-(q-1)/2..(q-1)/2]	11843	-11710	[-1149, 1656, -119, -619, 1646	[-1433, -1566, 466, 76, -724, -	12197	-11892	[1637, -1204, -1641, -98, -138	[-178, -1463, -709, -532, -758,	16790739	-16435708	[516, 334, -160, -1485, -1177,	[806, 1310, -1663, 1538, -368,	1881	-1885	[-1045, 811, 921, 1166, -1622,	[1551, 635, 216, 1579, 1444,
1000000	[-(q-1)/2..(q-1)/2]	12114	-12001	[1658, -304, -1509, -920, -120	[526, -1500, 1626, -879, -1385	12316	-12067	[1511, -154, -1349, -1307, -73	[-1629, -1598, 1370, 847, 147	17326746	-17432538	[-1036, -971, 421, 1615, 457, 5	[1060, 394, -1662, -477, 1207,	1886	-1890	[720, -52, -1188, 392, -1070, 8	[-812, -852, 1569, -1074, 722,
10000000	[-(q-1)/2..(q-1)/2]	12641	-12402	[1621, 81, -1568, 967, 1621, 977, 1617, 1566, -1051, 52, 856, -465, 1328, 1465, -1570, 202, -1386, -983, -431, -526, -1009, -81, -35, -1126, -389, -473, -76, 582, 892, -656, 1457, 1561, -420, -774, -1049, -614, -1609, 66, 1310, -1525, 1612, -208, 369, -94, 902, -1650, 367, 821, -76, -457, -1532, -1213, -889, -438, -854, -763, -678, 821, 1573, -1184, -632, 1006, -1651, -1063, -567, -1200, -479, 209, 255, -349, -837, 75, 760, 421, -1149, 960, -42, 81, -1147, 607, -434, -1586, -110, -636, 408, 955, 313, -119, -1586, -613, -1152, -801, -1307, 796, 870, -687, -49, -1488, -1281, 964, -423, -691, 1481, -784, -960, 1351, -1200, -709, -605, 394, -1063, 933, 1131, -1625, 1284, 730, -1368, 699, 114, -717, 1477, -572, 759, -1004, 195, -550, -305, -674, -1628, 364, 1481, -149, -1561, 575, -1204, -1357, -1511, 90, -1602, 914, 1330, -717, 364, -76, -1478, -308, 277, -1396, -448, 323, 1030, -1014, -646, 1664, 1215, 30, -29, -817, 1011, 870, -944, 1130, 1089, 645, -257, -1430, -866, 939, 1122, -1336, 1109, 250, 1324, 1014, 129, 820, -1240, -1505, 1019, 899, -470, -514, 1522, 1328, -319, -1632, -206, -1020, -1206, -1649, -602, 6, 1170, -833, -199, -395, -372, 582, -606, -1412, 1552, 353, -743, -1138, -515, -1218, -1007, -1333, 600, -681, 1244, 709, 1661, 799, 1422, 822, -560, -1573, -1659, 1498, -1368, -1384, 688, 1131, -1463, 1327, 1434, -187, -1525, -506, 1624, 902, 803, -52, 364, 1111, -1620, 1550, 504, -1158, -295, 866, -785, 1473, 492, -366, -909, -1167, 869, -990, 749, -1435, 1003, 817, 693, 1313]	[-1639, -1454, -535, 427, 118, 367, 827, 1086, 687, -205, -1627, -505, 1219, -1568, -939, -1618, -31, -552, 231, 1157, -1172, 497, -1462, -1023, -658, -485, -1168, -1033, -142, -531, -468, 406, -152, -701, -991, 814, -996, -1578, -658, -847, -1396, 826, -790, 1297, -1128, 652, -160, 298, 1235, 1131, -677, -853, -1095, 889, -603, 805, 264, -751, 1496, -25, -1554, -136, 1088, 1154, 1340, -197, -1606, 360, 915, -654, 1525, 1366, -1044, 1236, 1588, 1253, 1371, -934, 441, -941, -132, -176, -1122, -1452, -1488, 457, -475, 261, 1419, -814, 1535, -1541, 1159, -428, -460, 1059, 1507, 981, -160, 1254, 498, 1327, 1324, -824, -197, -963, 157, -562, -1411, 378, 1340, 536, -1400, -1186, -331, -862, -1087, 732, 90, -1185, -779, 241, 623, -666, -1487, -1001, -902, 482, 1387, 1588, -776, -1323, -440, 40, 565, -1371, 793, -765, 829, -60, -656, 1543, 37, 433, 1149, 1555, -1646, 1286, -1273, 1574, 433, 1252, -751, -1276, -92, 220, -1543, 671, 894, 759, -827, 1654, -1191, 1264, 1556, 181, -519, 870, -788, -473, -655, -752, -1269, -463, -1156, -715, -1574, -552, 166, -637, -90, 609, -1478, 611, 1625, -1470, 426, 920, -1105, 412, 624, 1229, 646, -482, 1624, -815, -1037, 435, -50, 451, 236, 659, 73, 864, -570, -911, 1059, -805, -1617, -751, 996, -491, 445, -416, 1361, 698, -1635, 869, -1366, 471, -1181, 1394, 913, -848, 1387, -1479, 1546, -508, 556, 1434, 1530, 1038, 965, 719, -1066, 955, 1238, -386, -1503, -1278, -1561, -233, -819, -107, -528, -1285, 222, -1111, -855, 766, -179, 1519, -1410, -386, -1286, 682]	12499	-12488	[1588, 789, 412, 412, 649, 1096, -648, -975, 1655, -650, 1469, 112, 649, 1494, -1602, -1210, 1039, 805, -78, 439, 1360, 802, -1033, -1046, 479, 921, -398, 1105, 1105, -1553, -1533, 35, -573, 1588, -24, 103, 602, 220, 506, 1265, 1575, 831, 964, 482, 574, -659, -1584, -297, -477, -253, 1044, 916, 1151, 253, -946, -263, 727, 168, -34, 688, -111, 910, -820, -808, -363, 1208, 1166, -963, -1480, -148, 521, -1485, -1377, -697, -262, -634, 433, 1050, -1451, -1381, -1405, 1585, -62, 521, 538, 998, 643, -1577, 1260, -1357, 578, -191, 808, 1345, -565, -605, -1319, -1378, 788, 964, -1551, 997, 649, 52, 980, -398, 1389, -136, 613, 653, -872, -685, 383, -222, -927, -101, 428, -1568, 188, -536, 647, 1284, 1273, -1525, -1622, 559, 997, 1303, -238, 407, -293, -1587, 425, 1495, 1445, -1409, 1372, -1303, 277, 1501, 415, -1148, 431, 1384, -151, -827, 515, -311, -657, -1245, 922, -945, 1362, -173, 243, 317, -1646, -855, -495, 1317, 761, 37, 1621, -81, -288, -561, -1340, 1526, 41, -1281, 1633, 309, -72, -1026, -751, -1502, 1036, 727, 1247, -504, -641, -758, -213, 363, -1548, -1001, 1521, -1083, 601, 416, 1267, 1078, 267, -1097, 860, -93, -1530, -1222, 712, -1589, 506, 1423, 1145, 1412, 82, -828, -875, 149, 1257, -1464, -444, 1203, 1396, -606, -1554, 407, -595, -150, 613, -1654, -1335, -90, 104, -1276, -1567, -612, 1142, -757, 1175, 1429, 330, 989, -39, 1455, -1351, 81, 232, 928, 794, -476, 1075, -1633, 510, 1611, -1602, -1194, -1023, -1051, 1003, 341, -514, -715, -319, -1182, -706, -1021]	[-447, -1620, 1223, 749, -1626, 246, 1651, -511, 1573, 1405, -1116, -510, 1473, -249, 551, -1498, 403, -414, 713, -1311, 1156, -117, -1031, 197, 1567, -28, -434, -1500, 1474, -685, 1513, -601, 531, 821, 34, 1234, -886, -1399, -277, 1549, 564, 1403, -861, -194, -1350, -222, -109, 427, -1646, 1474, -1251, -1389, 1641, 1099, 938, 1375, -1611, -1539, 820, 314, -459, -386, 838, -915, -383, -1150, 1547, 176, -1515, 316, 676, -1660, 430, 107, -1577, -855, -1059, -293, 805, 375, 459, -476, -64, -1199, 398, -280, 465, 720, -562, 47, -29, -60, 358, -1530, -1257, -311, -1144, 936, -438, -1366, 1178, 592, 1200, 831, 913, 1405, 825, 269, -1174, 634, 298, 928, -1200, 1548, 963, 785, 765, 480, -413, -1356, 405, 595, 183, 702, 342, -507, 1250, -245, 1477, 611, -1428, -1063, -1526, -260, 1605, 757, 1280, -1436, -1388, -781, 909, 173, -18, 1097, 1538, -517, 881, 1104, 1547, -1443, 169, 1354, -301, 781, -641, 1287, -1469, 108, -1222, -1426, -129, -464, -1050, 874, -802, 20, -1403, 607, -1635, -782, -1167, 8, 1366, -241, -740, -1429, 1108, 1273, 547, 856, 1117, 203, 996, -390, -1217, -1107, 25, -525, 1173, -684, 117, -524, -56, 366, 117, 302, -1074, 423, -1580, -431, 816, 33, 1454, 1027, 76, -1086, -1522, 733, 1021, 1213, 19, -1153, 164, 726, -343, -952, -1203, 977, 1335, 862, 1287, -1261, 761, 1497, 872, -129, -1565, 1363, 1488, -86, -239, -625, 305, -281, -713, 945, 816, 545, -670, -1361, 1086, -1138, 326, 883, -759, 747, 1292, -1197, -1290, 1542, -1193, 1305, 187, 1330, -1268, 54]	17971416	-17830932	[-151, 838, 1604, -1036, -511, 770, -586, 509, -595, 1260, 1199, -853, 1636, 1456, -958, 1105, -375, -42, 522, 1456, 207, -572, 675, -1610, 45, -201, 551, 211, 1273, 1480, 85, -256, -703, 1453, 132, -1280, -327, -1426, -1625, 759, 1238, -1292, 1531, -17, 1289, 1592, 1354, -1023, -423, -681, -926, 302, 1178, -659, 201, -240, -1335, 153, 873, 166, 311, 1638, 427, -431, 782, 782, 1056, 698, 243, 411, 1243, 520, 730, 511, -446, 561, 501, -1658, -736, 1120, 479, -1547, 709, 1625, -319, -763, -1425, -270, 840, 970, -984, -967, 1024, -1278, -792, 366, -1226, 1247, 1306, 960, -36, -1220, -840, -1350, -533, 12, -35, 827, 702, 1545, 1337, -324, 960, 1236, 1006, 606, -1262, 356, -858, -512, 990, -670, -1547, 678, 720, -974, 1359, 792, -1346, 492, -288, 723, -610, -1221, -1253, 108, -1218, 657, 1443, 71, -1501, 1266, -1571, -442, 933, 584, -1108, -353, 1628, 1487, -335, 1407, -759, -995, -1288, 482, -181, 966, -319, 1517, -1377, 1113, -259, 157, -140, 1414, 1150, -756, 1216, -723, -1544, -657, 1299, 1583, 1423, 880, 1657, 1162, -561, -1013, 45, -1069, 415, -1562, -84, 1005, 442, 665, 6, 1630, 136, -385, 384, 1000, 1408, 751, 599, 1271, -115, -406, 1393, 106, 324, 1278, -1041, 131, -963, 1379, 1048, 1505, 394, -854, -1285, 1545, 59, -347, -778, 1385, -429, 1645, -465, 1286, 887, -1204, 572, -957, -1411, -202, -1398, 1201, 317, -1343, -567, -1623, -1011, 1194, 1639, 0, 1061, -220, -299, -1375, -734, 1198, -1589, 177, -694, -776, -859, -183, 1539, 131, 1169, 1405, -315, 471]	[-811, 1115, 982, 1552, 1567, 65, -1178, 1337, -327, 421, 812, 649, 1100, 1454, -1030, -586, -1348, -856, 961, 1143, 1255, 1546, 722, -552, 1469, -1101, 1213, -983, -401, 453, 946, -691, -1173, 742, -516, 1514, 1636, -1426, -796, 1276, 1449, 1537, -932, 1416, -330, 273, 250, 1534, 487, -1649, 1026, 26, 551, -765, 30, 314, -763, -1359, -167, -89, 212, -654, -1229, 1485, 492, 194, 38, -1144, -693, -848, 237, -824, 172, 139, -126, 695, -103, -206, -1317, -1629, 657, 1182, 1236, -15, 342, 1492, 1029, 1245, -987, 1186, -386, 1535, 1290, -981, -211, -580, 1064, -811, -436, -648, -1627, -1624, 89, 1301, -852, -765, 187, -956, 491, -574, -1593, 1014, 880, 528, 1026, 929, -717, 200, -775, -559, 826, 724, -339, 238, 644, -644, 1065, 10, 562, -621, 1220, -24, -1445, -1019, -1215, 1133, -715, 1402, 992, -178, -1616, -916, -1061, 353, 514, -271, -1180, -1051, 268, -1243, 1201, 1368, 1584, -1190, -811, 1176, -222, 721, -421, -559, 1218, -695, -166, 501, 684, 1117, 537, 1007, -1002, 1033, -1441, -916, -367, 581, -1083, 522, -220, 90, 728, 1615, -1647, -700, 610, 1052, 1085, 1206, 140, -219, -46, 1188, 26, 1335, -500, -370, -576, -1199, 1082, -414, -275, -1642, -370, -813, -362, 816, -784, 941, 234, -311, -1080, 1611, 1135, 191, -494, -508, -1277, -892, 89, 1555, 519, -1356, -495, -1295, 671, -1464, -1446, -418, -909, 1100, -1290, -682, 1296, 1399, 1091, 513, 223, 797, -1634, 372, -453, -579, -1288, -237, -1337, -184, -1565, 144, -957, -1550, -807, 1658, -1282, 897, -1384, -1382, 855, 134]	1908	-1906	[-918, -830, 848, 1615, 1109, -1172, 229, -453, 152, 1561, -960, 671, 1206, 1180, -398, -837, 958, -331, -852, -564, 1537, -1018, 1071, -1193, -802, 753, 375, 1447, 1416, 619, -329, -13, -1070, 780, -287, 1485, 517, 768, 1139, 746, 314, 412, 77, 1259, -1082, -264, 221, -562, 1454, 599, -1152, 200, -1233, 334, -1557, 1618, 192, 471, 998, 671, -1132, 598, -955, -969, 480, 956, -146, -816, 1062, -827, -1140, 1636, -334, -1077, -1167, 368, -1045, -78, 1581, 1587, 803, 385, 1372, -895, 877, -142, 1143, -740, -928, 1382, -330, -1560, -591, -395, 98, -1455, 137, 110, 1327, 873, 1476, -1242, 131, -1355, 19, -1404, 1196, -690, 719, -546, -1063, -447, 1383, 77, -1457, 821, 1496, -1630, 1140, -234, -1207, 249, -441, -1277, 407, -277, 1256, -726, -1124, -19, 1281, -1316, 1553, -11, -824, -1522, 1139, -1648, 1455, 600, -1548, 150, 849, 1627, -799, 1127, -1137, -602, 1627, -98, -673, -1499, -977, 1587, -1012, -348, -536, -306, -337, -1052, -1554, 612, 1431, 1614, 1110, -1028, -821, 99, 902, -159, -1561, 886, 1646, 1462, 471, 262, 298, -127, -722, -219, -1030, -436, 1520, 845, 323, 947, 1446, -1417, -1448, -1127, -234, -702, -277, 844, -764, -1154, 572, 39, -1581, 1407, 717, -681, 1565, -376, -67, -730, -1272, -1245, -223, 1254, 1468, -197, -17, -368, 1279, -847, 1074, -1555, -1297, 1213, -460, 453, -1124, -1493, 1170, 547, -1282, -934, -1222, 1335, -1303, -112, -1443, -666, 827, 1530, -718, -1421, 706, -1485, -1605, -404, 367, -396, -961, -129, 174, 1257, -13, -181, -1134, -1177, -1438, 683, -535, 1327]	[-38, 1060, 1534, -1367, 1035, -1613, -77, -1012, -33, 1025, -1656, 292, -1505, 143, -217, -1110, 473, -589, 1579, -1100, -14, -670, 1265, 1622, 1379, 78, 906, -57, -1377, -1397, -524, -1633, -463, 596, 184, -1460, -1058, -487, 1576, 116, 298, -797, 1232, 1109, -1556, -637, -746, 1231, 64, -1151, -1593, -1508, 496, 376, -1202, -534, 918, -506, -11, 684, 126, -407, -1597, -1081, 1534, -1552, -165, 240, -1324, 1251, -1346, -701, -422, -926, -1084, 1336, 500, 245, 978, -1497, -1503, 695, 1126, 306, 560, -531, 1194, 970, -616, -19, 655, -1276, -293, -851, -860, 221, 124, 880, -791, 808, -496, -50, 516, -856, -1305, 1357, 1613, 1515, -269, -925, -282, 466, -483, -856, 462, 1297, 1443, -257, -109, -1309, 1052, 804, -1215, -1, 781, -1109, 434, 225, -1538, 586, -51, 40, 1081, 455, -1154, 243, -1636, -567, 1107, -1105, -123, 228, 1283, 1505, -776, 775, -282, 217, -13, 323, 796, 505, -1256, 1460, 1325, 1513, 1523, 268, 912, 131, -1289, -1149, 899, -1111, -1482, 357, -244, -306, 683, 1630, -1423, -1220, 138, 992, -578, 142, 1184, -569, -820, 439, -352, 712, 1288, 1522, 1575, 595, 1597, 589, 1393, 1199, -429, 1124, -592, -629, 260, 1506, -1053, 1244, -721, -1620, -256, -1504, 554, -1026, 312, -968, -184, -450, 48, -1634, 418, 837, -712, 279, -480, 791, 1367, 1605, 224, -1467, 1482, -1148, -1291, -120, 1376, -1409, -1194, 606, -620, 910, -390, -1271, 292, 1265, -45, -1281, 324, 1507, -1228, 1625, -877, 904, 957, 649, 1607, -392, 627, 1271, 167, 1086, -299, 373, 532, -433, -287, -1107]
		12641	-12402			12499	-12488			17971416	-17830932			1908	-1906		

Intervals		
Function	Input	Output
NTT	-q/2 < r[j] < q/2	-8(q/2) < r[j] < 8(q/2)
Montgomery	-1.8*10^7 < x < 1.8*10^7	-2(q/2) < y < 2(q/2)
Barrett	-8(q/2) < r[j] < 8(q/2)	-q/2 < r[j] < q/2
INV_NTT	-(q-1)/2..(q-1)/2	-((q-1)/2 + 15)..((q-1)/2 + 15)

q=3329
3q=9987
q/2=1664
8(q/2)=13312

input **INVTT** 10000 executions		output **INVTT** 10000 executions	
Max	Min	Max	Min
1664	-1664	1679	-1679

Bibliography

[1] Cisco. *Cisco Visual Networking Index: Forecast and Trends, 2018 2023*. https://www.cisco.com/c/en/us/solutions/collateral/executive-perspectives/annual-internet-report/white-paper-c11-741490.pdf (cit. on p. ii).

[2] Peter W. Shor. «Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer». In: *SIAM Journal on Computing* 26.5 (Oct. 1997), pp. 1484–1509. DOI: 10.1137/s0097539795293172. URL: https://epubs.siam.org/doi/10.1137/S0097539795293172 (cit. on p. 5).

[3] Lily Chen, Stephen Jordan, Yi-Kai Liu, Dustin Moody, Rene Peralta, Ray Perlner, and Daniel Smith-Tone. «NIST Released NISTIR 8105, Report on Post-Quantum Cryptography». In: (2016) (cit. on p. 5).

[4] Michael Nielsen and Isaac Chuang. *Quantum Computation and Quantum Information*. Cambridge: Cambridge University Press, 2000. ISBN: 0–521–63503–9 (cit. on p. 5).

[5] Public-key Cryptosystems, Oded Goldreich, Shafi Goldwasser, and Shai Halevi. «Public-Key Cryptosystems from Lattice Reduction Problems». In: (Dec. 1996) (cit. on p. 13).

[6] Phong Q. Nguyen. «Cryptanalysis of the Goldreich-Goldwasser-Halevi Cryptosystem from Crypto '97». In: *Advances in Cryptology - CRYPTO '99, 19th Annual International Cryptology Conference, Santa Barbara, California, USA, August 15-19, 1999, Proceedings*. Vol. 1666. Lecture Notes in Computer Science. Springer, 1999, pp. 288–304. DOI: 10.1007/3-540-48405-1_18 (cit. on p. 14).

[7] Oded Regev. «On Lattices, Learning with Errors, Random Linear Codes, and Cryptography». In: vol. 56. Jan. 2005, pp. 84–93. DOI: 10.1145/1568318.1568324 (cit. on pp. 15, 16).

[8] Jintai Ding, Xiang Xie, and Xiaodong Lin. *A Simple Provably Secure Key Exchange Scheme Based on the Learning with Errors Problem*. Cryptology ePrint Archive, Paper 2012/688. https://eprint.iacr.org/2012/688. 2012. URL: https://eprint.iacr.org/2012/688 (cit. on p. 22).

[9] Chris Peikert. *Lattice Cryptography for the Internet*. Cryptology ePrint Archive, Paper 2014/070. https://eprint.iacr.org/2014/070. 2014. URL: https://eprint.iacr.org/2014/070 (cit. on p. 22).

[10] «NIST Asks Public to Help Future-Proof Electronic Information». In: (2016) (cit. on p. 23).

[11] John M Pollard. «The fast Fourier transform in a finite field». In: *Mathematics of computation* 25.114 (1971), pp. 365–374 (cit. on p. 26).

[12] Ramesh Agarwal and Sidney Burrus. «Number theoretic transforms to implement fast digital convolution». In: *Proceedings of the IEEE* 63.4 (1975), pp. 550–560 (cit. on p. 26).

[13] Lawrence C. Washington. *Introduction to cyclotomic fields*. Graduate Texts in Mathematics 83, Springer-Verlag, 1997 (cit. on p. 27).

[14] Zhichuang Liang and Yunlei Zhao. *Number Theoretic Transform and Its Applications in Lattice-based Cryptosystems: A Survey*. 2022. arXiv: 2211.13546 [cs.CR] (cit. on p. 28).

[15] Franz Winkler. *Polynomial algorithms in computer algebra*. Wien: Springer-Verlag, 1996. ISBN: 3–211–82759–5 (cit. on p. 29).

[16] Gregor Seiler. *Faster AVX2 optimized NTT multiplication for Ring-LWE lattice cryptography*. Cryptology ePrint Archive, Paper 2018/039. https://eprint.iacr.org/2018/039. 2018. URL: https://eprint.iacr.org/2018/039 (cit. on p. 30).

[17] Daniel J. Bernstein. *Multidigit multiplication for mathematicians*. http://cr.yp.to/papers.html#m3. 2001 (cit. on p. 30).

[18] Roberto Avanzi et al. *CRYSTALS-Kyber (version 3.02) – Submission to round 3 of the NIST post-quantum project*. https://pq-crystals.org/kyber/data/kyber-specification-round3-20210804.pdf. 2021 (cit. on pp. 31, 38, 41–46).

[19] James W Cooley and John W Tukey. «An algorithm for the machine calculation of complex Fourier series». In: *Mathematics of computation* 19.90 (1965), pp. 297–301 (cit. on p. 32).

[20] W. Morven Gentleman and G. Sande. «Fast Fourier Transforms: for fun and profit». In: *AFIPS '66*. Vol. 29. 1966, pp. 563–578 (cit. on p. 33).

[21] ISO. *ISO/IEC 27005 2008*. Information technology Security techniques-Information security risk management" ISO/IEC FIDIS 27005 2008 (cit. on p. 47).

[22] MITRE. *Mitre*. https://www.mitre.org/our-impact/rd-centers/national-cybersecurity-ffrdc (cit. on p. 48).

[23] MITRE. *CVE List since 1999*. https://cve.mitre.org/ (cit. on p. 49).

[24] MITRE. *Weaknesses in Software Written in C*. https://cwe.mitre.org/data/definitions/658.html (cit. on p. 53).

[25] Microsoft. *Why Rust for safe systems programming*. https://msrc.microsoft.com/blog/2019/07/why-rust-for-safe-systems-programming/ (cit. on p. 55).

[26] NIST. *Safer Languages*. https://www.nist.gov/itl/ssd/software-quality-group/safer-languages (cit. on p. 64).

[27] Rust's Community. *Rust Embedded Book*. https://docs.rust-embedded.org/book/intro/hardware.html (cit. on p. 65).

[28] ST. *Datasheet STM32F303xB STM32F303xC*. https://www.st.com/resource/en/datasheet/stm32f303cb.pdf (cit. on p. 66).

[29] Elaine Barker (NIST), John Kelsey (NIST), Kerry McKay (NIST), Allen Roginsky (NIST), and Meltem Sönmez Turan (NIST). *Recommendation for Random Bit Generator (RBG) Constructions (3rd Draft)*. https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-90C.3pd.pdf (cit. on p. 67).

[30] 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*. Los Alamitos, CA, USA: IEEE Computer Society, 2022, pp. 19–30. DOI: 10.1109/EuroSPW55150.2022.00010. URL: https://eprint.iacr.org/2022/337 (cit. on pp. 67, 68).

[31] Matthias J. Kannwischer, Richard Petri, Joost Rijneveld, Peter Schwabe, and Ko Stoffelen. *PQM4: Post-quantum crypto library for the ARM Cortex-M4*. https://github.com/mupq/pqm4 (cit. on p. 68).

[32] PQM4. *Benchmarks*. https://github.com/mupq/pqm4/blob/master/benchmarks.csv (cit. on p. 68).

[33] Matthias J. Kannwischer, Joost Rijneveld, Peter Schwabe, and Ko Stoffelen. *pqm4: Testing and Benchmarking NIST PQC on ARM Cortex-M4*. https://csrc.nist.gov/CSRC/media/Events/Second-PQC-Standardization-Conference/documents/accepted-papers/kannwischer-pqm4.pdf (cit. on p. 68).

[34] National Institute of Standards and Technology. *FIPS PUB 140-2 SECURITY REQUIREMENTS FOR CRYPTOGRAPHIC MODULES*. https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.140-2.pdf. May 25, 2001 (cit. on p. 77).

[35] ST. *UM1472 User manual - Discovery kit with STM32F407VG MCU*. https://www.st.com/resource/en/user_manual/um1472-discovery-kit-with-stm32f407vg-mcu-stmicroelectronics.pdf (cit. on p. 79).

[36] ST. *RM0090 Reference manual*. https://www.st.com/resource/en/reference_manual/dm00031020-stm32f405-415-stm32f407-417-stm32f427-437-and-stm32f429-439-advanced-arm-based-32-bit-mcus-stmicroelectronics.pdf (cit. on pp. 78, 80, 95).

[37] ST. *AN4230 Introduction to random number generation validation using the NIST statistical test suite for STM32 MCUs and MPUs*. https://www.st.com/resource/en/application_note/an4230-introduction-to-random-number-generation-validation-using-the-nist-statistical-test-suite-for-stm32-mcus-and-mpus-stmicroelectronics.pdf (cit. on p. 80).