POLITECNICO DI TORINO

Master's Degree in Software engineering

Master's Degree Thesis

Radiation Hardened Rust Implementation of a Crystals-Kyber Encryption Scheme for Space Applications

Supervisors

Prof. RICCARDO SISTO

Dr. FEREYDOUN DANESHGARAN

Dr. MARINA MONDIN

Candidate

FEDERICO SANFRANCESCO

APRIL 2025

Abstract

It is a well-known fact that the security of traditional cryptographic techniques relies on the algorithmic complexity of solving certain mathematical problems, such as one-way functions and factorization algorithms. However, the advent of quantum computers, which can solve factorization problems at unprecedented speeds, poses a significant threat to this security paradigm. This has created an urgent need for alternative cryptographic solutions. Two primary approaches have emerged: Quantum Cryptography, which ensures security by modifying the physical transmission of cryptographic information according to fundamental physical principles, and Post-Quantum Cryptography, which adapts algorithmic processing to use techniques resistant to quantum computing attacks. Among these, the Crystals Kyber algorithm has been recognized as a candidate for new cryptographic standards. This study specifically explored the integration of Post-Quantum Cryptography within a satellite communication scenario. The project involved the design and implementation of a Crystals Kyber-based encryption scheme and an information protection mechanism leveraging error correction through Turbo codes. Both components were developed and validated using the Rust programming language, contributing to a secure communication framework suitable for future satellite networks.

To the fools who dream

Table of Contents

List of Tables

List of Figures

Acronyms

AWGN Additive White Gaussian Noise.

BCH Bose–Chaudhuri–Hocquenghem codes.

BER Bit Error Rate.

BKZ Block Korkine-Zolotarev Algorithm.

BSC Binary Symmetric Channel.

CPA Chosen-Plaintext Attack.

CPAPKE CPA-Secure Public Key Encryption.

CRQC Cryptographically-Relevant Quantum Computer.

CVP Closest Vector Problem.

DEP Data Execution Prevention.

DRAM Dynamic Random Access Memory.

ECSS European Cooperation for Space Standardization.

FEC Forward Error Correction.

FFI Foreign Function Interface.

FO Fujisaki–Okamoto.

FSP Finite State Permuter.

HAL Hardware Abstraction Layer.

IND-CCA2 Indistinguishability under Chosen-Ciphertext Attack.

KEM Key Encapsulation Mechanism.

LDPC Low-Density Parity-Check codes.

LEO Low Earth Orbit.

LET Linear Energy Transfer.

LFSR Linear Feedback Shift Register.

LLR Log-Likelihood Ratio.

LWE Learning With Errors.

MAP Maximum A Posteriori Probability.

MBU Multiple Bit Upset.

MLWE Module Learning With Errors.

NIST National Institute of Standards and Technology.

PCCC Parallel Concatenated Convolutional Codes.

PKE Public Key Encryption.

QROM Quantum Random Oracle Model.

RAII Resource Acquisition Is Initialization.

ROP Return-Oriented Programming.

RSA Rivest-Shamir-Adleman.

RSC Recursive Systematic Convolutional.

RTOS Real-Time Operating System.

SEE Single Event Effects.

SEH Structured Exception Handling.

SEU Single Event Upset.

SISO Soft Input Soft Output.

SIVP Shortest Independent Vector Problem.

SNR Signal-to-Noise Ratio.

SRAM Static Random Access Memory.

SVP Shortest Vector Problem.

TID Total Ionizing Dose.

TLS Transport Layer Security.

Chapter 1

Introduction

1.1 Thesis Objectives

The primary objective of this project is to develop an efficient software implementation of the Crystals-Kyber cryptographic algorithm, simulating its application in the space domain under non-ideal transmission conditions. This was achieved using Rust, a programming language designed to ensure both high efficiency and robust security. The project represents a preliminary approach to adapt high-level requirements from space applications to the new challenges posed by emerging technologies. Performance evaluations were conducted to compare the results obtained with data available from the official documentation of well-known, thoroughly tested, and widely used algorithms in the industry. Since current validation protocols employed by organizations such as NASA and ESA are highly rigorous and require years before becoming de facto standards, they primarily use simple, reliable architectures and programming languages like C, which have a proven track record of dependability in critical environments. However, this project sought to explore the potential of modern approaches, attempting to bridge the gap between current practices and future needs. Using advanced features of Rust and the most recent algorithms, the work aims to make this anticipated future more tangible, providing a glimpse into how emerging technologies might reshape the landscape of space applications.

1.2 Context

The transmission of data via satellite is a crucial and extremely important method of communication and data transmission all over the globe. However, satellite systems are exposed to some of the most extreme conditions on Earth, making data increasingly susceptible to corruption due to various disruptive factors. For

example, data can be compromised during the encryption process because of Single Event Upsets SEU and Multiple Event Upsets MBU caused by radiation, or during transmission due to channel noise. Encryption and decryption are common security operations handled by every system that ensures data integrity and/or privacy, from earth to space and vice versa. For satellites, employing the most advanced cryptographic methods is essential to safeguard data from alterations during transmission or while stored. Cryptography and fault detection are the key to addressing these challenges, ensuring the confidentiality of information and securing the data transmission process. There are numerous hardware and software solutions designed to mitigate these vulnerabilities, each employing different strategies to distribute the encryption keys necessary to protect and restore the data [1].

1.2.1 Space Radiations

One of the main concerns about space data transmissions is represented by radiations. The high presence of this natural phenomenon makes communication in one of the harshest environments a real challenge, when dealing with noisy channels and sensitive hardware. This natural radiation includes electrons and protons trapped within planetary magnetic fields (e.g., Earth, Jupiter), high-energy protons and heavier nuclei emitted during solar events, and cosmic rays originating from supernova explosions both within and beyond our galaxy. Inside spacecraft such as the International Space Station, primary cosmic rays, interact with the surrounding material, producing secondary neutrons. These neutrons can contribute to Single Event Effects SEE in electronic systems, adding complexity to space radiation challenges.

The radiation dose in space is relatively low, ranging from 10^{-4} to 10^{-2} rad/s. However, extended mission durations, often measured in years, result in significant cumulative doses, reaching total ionizing dose TID levels of 10^5 rad or more. This requires strict characterization and qualification of electronic devices to meet mission-specific requirements.

Radiation effects are described using parameters like stopping power or linear energy transfer LET, which quantifies the energy deposited per unit length as charged particles pass through matter. The LET depends on factors such as the energy, mass and charge of the particle, as well as the material properties of the target. For example, in silicon, an absorbed dose of 100 erg per gram corresponds to one rad, while the equivalent SI is gray (1 gray = 100 rad).

1.2.2 Single-Event Effects (SEEs)

The possibility of single event upsets was first postulated in 1962 by Wallmark and Marcus [2]. SEEs occur when a charged particle deposits enough energy

in an electronic device to exceed a critical charge threshold (Q_{crit}), causing a transient or permanent malfunction. SEEs can be classified as either destructive, leading to device failure, or nondestructive, resulting in temporary loss of data or functionality. The mechanism behind SEEs involves energy loss through ionization as a particle traverses the device. This process generates charge carriers (electron hole pairs), the number of pairs proportional to the stopping power of the particle. For example, in silicon, 22.5 MeV of deposited energy generates one picocoulomb (pC) of charge. The collected charge may recombine or accumulate at sensitive circuit nodes, triggering an SEE if Q_{crit} is exceeded. The SEE sensitivity test involves controlled experiments using particle accelerators. By varying particle mass, energy, and angle of incidence, researchers measure SEE rates as a function of LET and cross-sectional area. These results are combined with spacecraft trajectory data to estimate mission-specific SEE rates. Usually, particles entering sensitive regions at oblique angles can deposit significantly more energy, effectively doubling the LET at 60° incidence compared to normal angles. SEEs is a broad category that includes all types of interactions in electronic components caused by a single high-energy particle (e.g., a cosmic ray or a proton). Single-Event Effects (SEEs) can be categorized into destructive and nondestructive types. For the purposes of this project, the analysis was specifically limited to nondestructive effects, with a particular emphasis on Single-Event Upsets (SEUs) and Multiple-Bit Upsets (MBUs), as these were deemed to be the most relevant phenomena.

SEUs and MBUs

Single-Event Upsets (SEUs) refer to the alteration of the state of bistable elements, such as flip-flops or memory cells, caused by the impact of energetic particles like heavy ions or protons. These effects are nondestructive and can typically be corrected by rewriting the affected element. SEUs occur when a particle strike introduces sufficient charge to exceed a circuit node's critical charge Q_{crit}, leading to a change in the logic state, commonly referred to as a "bit flip". The susceptibility of a device to SEUs is determined by two key parameters: the threshold linear energy transfer (LET), representing the minimum energy needed to cause an upset, and the saturation cross-section, which depends on the surface area of SEU-sensitive nodes. Memory technologies such as Static Random Access Memory SRAM and Dynamic Random Access Memory DRAM are particularly sensitive to SEUs. DRAM cells also require periodic refreshing to retain data, making them more prone to disturbances.

A **Multiple Bit Upset (MBU)** is a radiation-induced phenomenon in which multiple bits within a memory system or a digital circuit are simultaneously altered due to a single ionizing particle interaction. Unlike a Single Event Upset (SEU), which affects only one bit, MBUs typically occur when bits located in close physical

proximity experience correlated errors, often due to charge deposition across multiple nodes. MBUs still represent a significant challenge in high-reliability systems, as they can compromise data integrity and are more difficult to mitigate using conventional error correction techniques. The likelihood of MBUs increases with higher radiation levels, reduced feature sizes in integrated circuits, and increased memory density.

1.3 Realistic Simulation

Testing and simulating the effects of radiation, particularly in space applications, requires a meticulous and comprehensive approach that encompasses both hardware and software considerations. As highlighted in a study [3] from 2019 focused on the Single Event Effects (SEE) characterization of the AD9361 RF transceiver under proton irradiation, the ability to replicate space-like environmental conditions is essential to validate device performance in harsh environments. This research, conducted by the German Aerospace Center, outlines the complexities involved in simulating factors such as altitude, inclination angles, temperature variations, and shielding effects, which directly influence radiation exposure. For instance, the study investigates radiation conditions in Low Earth Orbit LEO using the **NASA AP-8** and **CREME96** models, which predict the flux of protons and cosmic rays at varying altitudes (e.g., 400 km and 800 km) and inclinations (e.g., 0°, 51.64°, and 98°). To achieve meaningful results, the device was tested under a controlled spectrum of proton energies, ranging from 4 MeV to 184 MeV, with fluence (fluence = flux * time) levels tailored to replicate the reference mission profiles. While precise replication is critical, strategic approximations, such as the adoption of a discrete set of proton energies and limited total fluence, allow for early-phase assessments of the device's robustness and susceptibility to SEE. The research demonstrates that a balanced approach combining accurate environmental replication with practical approximations is vital for advancing space-qualified technologies. Moreover, the inclination of the device relative to the incident radiation, as previously specified, influences error correlations. Under different conditions, such as inclination, the likelihood of Multiple Bit Upsets (MBUs) increases, causing several bits within a localized region to be affected simultaneously. This phenomenon leads to a higher degree of error correlation, making it challenging to accurately replicate in simulations without real-world test cases. To address this complexity, Turbo codes will be employed, considering an input scenario where errors are completely *uncorrelated*. This approach allows for a more precise evaluation of turbo codes' performance and potential, ensuring a clearer understanding of their effectiveness in mitigating errors under varying conditions.

Based on this premise, several cases have been analyzed in order to use some

valid approximations in the discussed implementation. This study [4], presented at CERN, has been particularly interesting as framework when quantifying the impact of radiation through analytical and experimental techniques, focusing on the calculation of error rates in radiation environments and their implications for space systems.

In this context, the error rate calculation is critical for predicting the reliability of electronic components in space. The sensitivity of a circuit to SEU or any SEE is characterized by its cross-section σ, which represents the effective area where radiation interactions can induce an error. The cross-section is determined experimentally by exposing the device to a particle beam and measuring the number of errors N observed for a given particle fluence Φ, as follows:

$$\sigma = \frac{N}{\Phi},$$

where Φ is the product of the particle flux ϕ (particles per unit area per second) and the exposure time t. Once the cross-section is known, the error rate in a specific radiation environment can be estimated by multiplying σ by the expected flux ϕ:

$$\text{Error Rate} = \sigma \cdot \phi.$$

For example, an SRAM irradiated with a proton beam of 100 MeV and flux $\phi = 10^5\,\text{p/cm}^2\text{/s}$ might exhibit $N = 1000$ errors for a total fluence of $\Phi = 10^{12}\,\text{p/cm}^2$. Using the formula above, the cross-section is calculated as:

$$\sigma = \frac{1000}{10^{12}} = 10^{-9}\,\text{cm}^2.$$

In an operational space environment with a similar flux, the error rate can be estimated as:

$$\text{Error Rate} = 10^{-9} \cdot 10^5 = 10^{-4}\,\text{errors/s}.$$

This approach is directly applicable to space systems, where understanding the error rate helps design radiation-hardened devices and implement error mitigation strategies, such as redundancy or error correction codes. From NASA's technical reports archive (NTRS) [5], various node-to-node communication architectures for space missions can be evaluated, with a particular focus on Earth vicinity communications. Among the proposed configurations, these specific architectures allow for 10 Mbps data transmission utilizing the Ka and X frequency bands: the link between a LEO satellite and a GEO satellite, the reverse communication from GEO to LEO, and the connection between a LEO satellite and a Shuttle. These configurations are particularly relevant for applications requiring secure and reliable data transmission, such as encrypted communications and secure telemetry exchange. The selection of LEO-GEO communication links is motivated

by their ability to provide global coverage with extended operational time, ensuring continuous connectivity between spaceborne assets and ground stations. The LEO-to-LEO and LEO-to-Shuttle links, on the other hand, are characterized by their low-latency transmission, making them suitable for time-sensitive data exchanges that require minimal propagation delay. The use of the **Ka-band** (26–40 GHz) is advantageous due to its high bandwidth and superior data rates, which are essential for transmitting encrypted data efficiently. However, since Ka-band is susceptible to atmospheric attenuation, particularly in adverse weather conditions, it is complemented by the **X-band** (7–8 GHz), which provides greater resilience to signal degradation and has a long-standing history of use in military and space applications. This dual-band approach ensures both high data throughput and robust signal integrity, reducing the risk of interference or data corruption during transmission. By leveraging these frequency bands and orbital configurations, the communication links analyzed in the article offer a balance between security, reliability, and efficiency, making them well-suited for space missions requiring protected data transmission and robust communication infrastructure. In order to evaluate the efficiency of a transmission algorithm in the aforementioned simulation, a data rate of $R = 10$ Mbps and a bit error rate (BER) of 10^{-4} are considered. The total number of bits transmitted per second is given by:

$$N_{\text{total}} = R \times 1s = 10^7 \text{ bits/s} \tag{1.1}$$

Given the BER, the expected number of bit errors per second can be estimated as:

$$N_{\text{errors}} = \text{BER} \times N_{\text{total}} = 10^{-4} \times 10^7 = 10^3 \tag{1.2}$$

This implies that, on average, 1000 erroneous bits occur every second. To assess the robustness of an error correction algorithm, a simulation can be performed by introducing errors at this rate and measuring the percentage of successfully recovered data. The efficiency η of the algorithm can be defined as:

$$\eta = \frac{N_{\text{corrected}}}{N_{\text{errors}}} \times 100\% \tag{1.3}$$

where $N_{\text{corrected}}$ represents the number of errors successfully corrected. By varying the BER and observing the impact on η, it is possible to determine the algorithm's reliability under different transmission conditions. The algorithm's efficiency will be further inspected in the next chapters.

1.4 Quantum Computing

Modern (classical) computers struggle with some specific complex tasks, like solving mathematical problems, forecasting the weather or modeling economies. We should however recall that this is also the reason why the cryptographic algorithms currently widely employed are (still) protecting us from cyber-attacks. Certain complex problems (like factorization, which is at the core of current cryptographic techniques) are (still) hard to solve for classical computers, because they are inefficient by design. They work serially, which means they handle one task at a time. Although parallel computing, which uses multiple processors to work on different tasks simultaneously, has been explored, progress has been slow because traditional processors are built for serial processing. True parallelism, performing many tasks at once, is a feature of quantum computers. In quantum computing, the smallest unit of information is called a qubit, which differs from a classical computer's bit. While a classical bit can only be in one of two states (either 0 or 1), a qubit can exist in a combination, or superposition, of both states at once. This means a qubit can represent 0, 1, or both simultaneously, with a specific probability for each state.

The quantum mechanical system taken into account must be the **qubit**. By definition [6], a **qubit** has a two-dimensional state space. Suppose $|0\rangle$ and $|1\rangle$ form an orthonormal basis for that state space. Then an arbitrary state vector in the state space can be written as:

$$|\psi\rangle = a|0\rangle + b|1\rangle, \tag{1.4}$$

where a and b are complex numbers. The condition that $|\psi\rangle$ be a unit vector, $\langle\psi|\psi\rangle = 1$, is therefore equivalent to:

$$|a|^2 + |b|^2 = 1. \tag{1.5}$$

The condition $\langle\psi|\psi\rangle = 1$ is often known as the **normalization condition** for state vectors and it stems from Bohr's conception of Copenhagen interpretation.

In quantum mechanic, **superpositions** of these two states, of the form $a|0\rangle+b|1\rangle$, can also exist, in which it is not possible to say that the qubit is definitely in the state $|0\rangle$, or definitely in the state $|1\rangle$.

Hence, any linear combination $\sum_i \alpha_i|\psi_i\rangle$ is a **superposition** of the states $|\psi_i\rangle$ with amplitude α_i for the state $|\psi_i\rangle$. For example, the state:

$$\frac{|0\rangle - |1\rangle}{\sqrt{2}} \tag{1.6}$$

is a superposition of the states $|0\rangle$ and $|1\rangle$ with amplitude $\frac{1}{\sqrt{2}}$ for the state $|0\rangle$, and amplitude $-\frac{1}{\sqrt{2}}$ for the state $|1\rangle$.

1.4.1 Quantum Threat

Quantum computers pose a significant threat to current cryptographic systems due to their ability to perform many computations in parallel far beyond the capability of traditional systems. Algorithms such as **Shor**'s (1994) could break widely used encryption methods like RSA, while **Grover**'s (1996) algorithm could weaken symmetric-key cryptography by reducing the complexity of brute force attacks. This quantum threat has implications for the security of cyber-systems, potentially compromising both the confidentiality and trust within these systems. To counteract this threat, quantum-resistant cryptographic techniques are being developed, which may include both classical and quantum-based solutions. The National Institute of Standards and Technology NIST has already made progress in standardizing post-quantum cryptographic algorithms and Crystals-Kyber successfully passed round 3 submission. However, the transition to quantum-safe cryptography is complex, requiring a well-coordinated development of new hardware and software, the establishment of standards, and the migration from legacy systems. This transition must be planned carefully to ensure cybersecurity is not compromised during the shift to quantum-safe systems. To determine the urgency of transitioning to quantum-safe cryptography, three critical parameters must be considered:

- **Tshelf-life:** The number of years for which the information should be protected.
- **Tmigration:** The time required to migrate the system to quantum-safe solutions.
- **Tthreat:** The timeline for when quantum computers will be capable of breaking current cryptographic systems.

The Mosca Inequality (2013) suggests that if the combined time required for migration and the desired information protection period exceeds the time before the quantum threat becomes concrete (TTHREAT), organizations may struggle to secure their systems. The Mosca Inequality can be written as:

$$\text{Tshelf-life} + \text{Tmigration} \leq \text{Tthreat} \tag{1.7}$$

The concept of a "Harvest Now, Decrypt Later" attack indicates that adversaries could intercept and store encrypted data now and decrypt it once a Cryptographically Relevant Quantum Computer CRQC becomes available. However, a rushed transition to quantum-safe systems could introduce vulnerabilities, especially in terms of interoperability and potential design flaws. The threat timeline (Tthreat) is currently difficult to assess, given the many scientific and engineering challenges in developing a quantum computer that can break modern cryptography. While

current obstacles suggest that CRQCs are still years away, rapid advances in quantum computing, fueled by substantial investments from governments and private entities, may accelerate this timeline, increasing the urgency for a well-planned transition to post-quantum cryptographic systems.

1.5 Reliable Data Transmission in Space: Challenges and Solutions

In order to enhance fault tolerance in space systems, various techniques can be employed. As part of the "hardening" process, radiation-hardened electronics are often shielded in a layer of depleted boron and mounted on insulating substrates, rather than on standard semiconductor wafers. This design enables them to endure far higher levels of radiation compared to commercial-grade chips. Radiation-hardened electronics, also called rad-hard electronics, are electronic components (circuits, transistors, resistors, diodes, capacitors, etc.), single-board computer CPUs, and sensors that are designed and produced to be less susceptible to damage from exposure to radiation and extreme temperatures (-55°C to 125°C). The main concern of this project, though, is focusing on logic and software damages and how to build a safe and reliable communication system.

1.5.1 Error Detection and Correction Techniques

Error detection and correction (EDAC) techniques are essential for ensuring the integrity of digital data during transmission or storage. Error Detection involves using additional information, like a checksum, to the data. This checksum allows the receiver or storage system to verify data correctness. If the checksum doesn't match, an error has occurred, prompting retransmission or correction. Error Correction incorporates redundancy into the data, which helps reconstruct the original data in case of errors. Techniques such as Parity Checking, Hamming Codes, Reed-Solomon, LDPC, Turbo Codes are widely employed in many fields. Turbo codes algorithm was taken into account in this project, to perform a safe communication of Crystals-Kyber's byte arrays. **Forward Error Correction** FEC is a category that implies adding redundancy bits to the data, enabling the system to reconstruct the original data in case of errors. Common FEC codes include convolutional codes and block codes. **Convolutional codes** are capable of processing the data into streams of bits and add check bits based on current and previous data and predefined coefficients. These are particularly effective in environments with high error rates. **Turbo codes** exploit the parallelism of multiple convolutional codes applied to bit streams in order to generate information in the form of coefficients and iteratively improve error correction through a soft-decoding approach, that

will be further inspected in the next chapters.
The structure of the document is organized as follows. Chapter 2 provides a detailed overview of the Rust programming language main features, including some projects fostering its applications in space. In Chapter 3, the CRYSTALS-Kyber algorithm is presented, exploring the mathematical aspects on which the algorithm relies. Chapter 4 focuses on the mechanisms and structures that Turbo Codes implement, from the available literature. Chapter 5 illustrates the structure of the software implementation in Rust of the previous illustrated concepts, summarizing the main strategies and the modular approach. Chapter 6 shows the results obtained by simulations. Chapter 7 concludes the document, summarizing the main findings and providing recommendations for future research.

Chapter 2

Rust Programming Language

2.1 Introduction

Rust is a modern systems programming language [7] designed to address critical challenges in software development, such as memory safety, concurrency, and performance. It was originally developed by Graydon Hoare in 2006 and later gained significant momentum under Mozilla Research's sponsorship. The language was officially introduced in 2010, with the first stable release, Rust 1.0, arriving in 2015. The primary motivation behind Rust's creation was to overcome the inherent safety and reliability issues often encountered in traditional low-level programming languages like C and C++. Rust achieves this by providing strong guarantees for memory safety and concurrency, enforced at compile time, without sacrificing performance or requiring a garbage collector. This approach allows developers to write efficient, low-level code while avoiding common pitfalls such as null pointer dereferencing, buffer overflows, and data races.

Rust was initially developed to improve the security and performance of Mozilla's projects, most notably the Firefox browser. Components like the Servo web rendering engine were built using Rust to demonstrate its capabilities in creating fast, secure, and reliable software. The language's emphasis on zero-cost abstractions and strict compile-time error checking aligns it with the needs of industries that demand high-performance, low-latency applications, including operating systems, embedded systems, and real-time systems. From the beginning, Rust's development has been driven by an open-source community, with a strong focus on collaboration and innovation. Its integrated tooling, including the Cargo build system and package manager, ensures a seamless development experience, fostering widespread adoption across various domains. Rust has since become a key tool for developers

seeking to build software that combines low-level control with high-level safety guarantees. The Rust Standard Library [7] will be used as a reference in the following sections as the language is analyzed in more depth.

2.2 Main Features

Rust is a statically typed programming language specifically designed to optimize both performance and safety, with a strong emphasis on secure concurrency and efficient memory management. Its syntax is similar to that of C++. Since 2021, the Rust Foundation has overseen its development, ensuring its continuous evolution and adoption. While modern C++ has introduced mechanisms such as *smart pointers* to enhance memory safety, several issues remain unresolved. A notable example is the "use-after-free" error, which occurs when a program accesses a pointer after the associated memory has been deallocated; for instance, invoking a lambda function after its captured references have been freed. Rust mitigates such issues through its ownership model and borrow checker, integral components of the compiler that enforce strict reference management rules. These mechanisms ensure that references cannot outlive the data they point to, thereby preventing memory violations at compile time and eliminating the need for *garbage collection*. Furthermore, Rust introduces the concept of *lifetimes*, which define the validity scope of references, preventing the use of dangling references (a persistent issue in C and C++). The significance of robust memory management becomes evident when considering security vulnerabilities.

2.2.1 Security Vulnerabilities Based on Unsafe Memory Access

Over the past decade, approximately 70% of security-related bugs in Microsoft products have been attributed to memory safety issues, a figure that is similarly reported for Google Chrome. In cybersecurity, various attacks exploit memory vulnerabilities in low-level programming languages such as C and C++, which lack strict memory safety guarantees. Memory vulnerabilities are often exploited by attackers to steal sensitive information, such as passwords, financial data, or intellectual property.

According to IBM's data breach report in 2024, "the global average cost of a data breach is estimated to be $4.45 million" [9]. Some vulnerabilities can lead to significantly higher costs. For instance, the Heartbleed vulnerability in OpenSSL is estimated to have caused damages exceeding $500 million. Data breaches can also involve multi-stage attacks that move from an initial entry point to deep inside a company's network. A notable example is Operation Soft Cell, where attackers

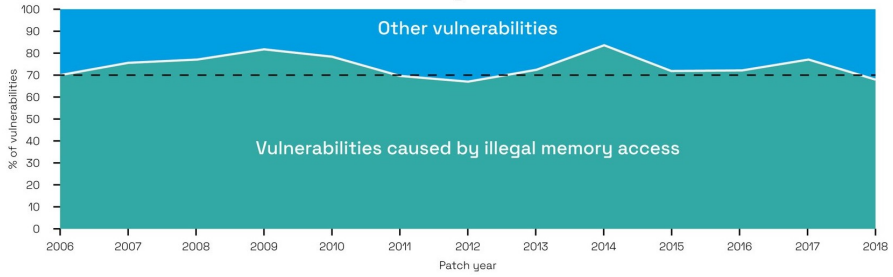


Figure 2.1: As the chart above illustrates, memory safety issues have remained dominant for over a decade. Source: [8]

were able to gain access to the web-facing servers of telecom companies through a code injection attack facilitated by a buffer overflow. This allowed them to steal account credentials, which were then used to create high-privilege user accounts. The most common types of memory vulnerabilities include:

- **Buffer overflow** and **heap overflow**: These attacks occur when a program writes more data than allocated into a memory buffer, leading to the corruption of adjacent memory locations. This vulnerability can allow attackers to overwrite control structures, alter execution flow, or execute arbitrary code. Buffer overflow typically affects stack memory, while heap overflow targets dynamically allocated memory.

- **Integer overflow**: This occurs when an arithmetic operation results in a value exceeding the maximum representable value of a data type, causing wraparound behavior. **Integer underflow**, conversely, happens when the result is lower than the minimum representable value. Attackers can exploit these vulnerabilities to manipulate memory allocation sizes, bypass security checks, or trigger unintended program behaviors.

- **Pointer subversion**: This involves the manipulation of pointers to modify critical memory regions, potentially allowing attackers to alter program behavior. This includes overwriting function pointers, modifying virtual table pointers (vtable hijacking), or corrupting structured exception handling SEH mechanisms. Such techniques can lead to arbitrary code execution.

- **Return-Oriented Programming** ROP: This advanced exploitation technique circumvents security mechanisms like Data Execution Prevention DEP. Instead of injecting new code, attackers use existing instruction sequences (gadgets) within legitimate code to chain together malicious operations, effectively hijacking program execution.

These vulnerabilities demonstrate the risks associated with manual memory management in languages like C and C++. Rust mitigates such issues through its ownership model and *borrow checker*, ensuring memory safety at compile time. Rust offers two distinct modes of programming: Safe Rust and Unsafe Rust. Safe Rust imposes constraints on developers, such as ownership and borrowing rules, ensuring memory safety and preventing undefined behavior. Conversely, Unsafe Rust provides lower-level access, including operations on raw pointers, thereby offering greater flexibility but also increasing the risk of memory errors. To mitigate potential issues, developers can encapsulate unsafe operations within high-level abstractions that guarantee safety in their usage. Unlike C++, where unsafe code may only manifest as a failure or security vulnerability at runtime, Rust's approach ensures that potential risks are identified and addressed at compile time. Rust's dual-mode system represents a significant advantage over traditional languages like C++, providing a balance between strict safety guarantees and the ability to perform low-level operations when necessary. This design philosophy not only enhances software reliability but also improves security, making Rust an increasingly preferred choice for system-level programming.

2.2.2 Common Programming Concepts

Borrowing and Ownership

Rust's borrow checker is a core mechanism that enforces ownership rules to ensure memory safety without relying on garbage collection. By preventing issues such as use-after-free errors, dangling pointers, and data races, this system guarantees safe memory access while maintaining performance comparable to manual memory management. Rust's approach enables deterministic memory deallocation, eliminating the runtime overhead typically associated with traditional garbage collection. Various programming languages adopt different memory management strategies. In garbage-collected languages such as Python, JavaScript, and C#, memory deallocation occurs automatically at runtime, simplifying development but introducing execution overhead and unpredictable pauses. In contrast, C and C++ rely on manual memory management, providing fine-grained control but increasing the risk of memory leaks and undefined behavior. Rust introduces an alternative paradigm through its ownership model, where the borrow checker enforces memory safety at compile time, preventing invalid memory access and optimizing resource utilization.

Memory management in Rust involves both stack and heap allocation. Stack memory is used for storing fixed-size data with well-defined lifetimes, allowing for fast access and automatic deallocation when a function scope terminates. Heap memory, on the other hand, accommodates dynamically allocated data, requiring explicit ownership handling to ensure proper deallocation. Rust's ownership model

dictates when memory is allocated and deallocated, enforcing safety guarantees without requiring runtime intervention.

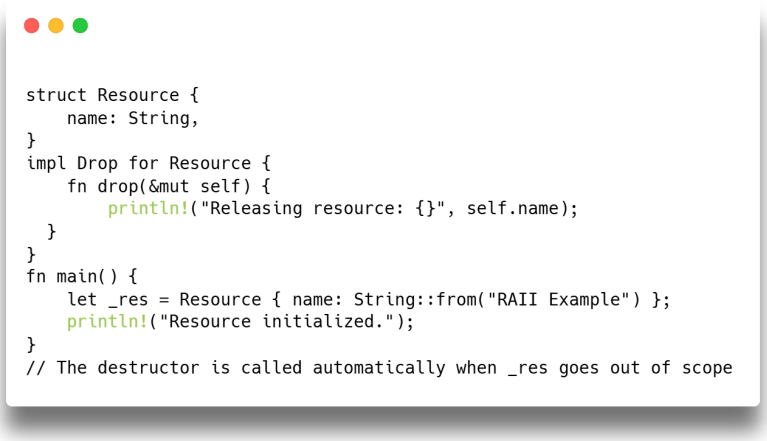
Ownership in Rust also governs how values are transferred between functions and scopes. A value may be moved, transferring ownership and invalidating the original reference, or cloned, creating a deep copy at the expense of additional memory usage. Borrowing allows references to be passed without transferring ownership, improving efficiency. Borrowing can be either immutable, permitting multiple concurrent references without modification, or mutable, which restricts access to a single reference at a time to prevent data races and maintain safety guarantees.

Rust's borrowing rules extend to concurrent programming by ensuring the absence of data races at compile time. Shared data structures, such as *mutexes*, enable safe concurrent access by enforcing exclusive access to mutable data when necessary. This mechanism ensures that multiple threads cannot simultaneously modify a shared resource in an unsafe manner. By enforcing these constraints, the borrow checker allows Rust to achieve memory safety without requiring garbage collection or runtime checks.

RAII Paradigm

Resource Acquisition Is Initialization RAII is a key programming paradigm originally introduced in C++, which ensures automatic and deterministic resource management. This concept revolves around the idea that resources, such as memory or file handles, are acquired during the initialization of objects and released when they go out of scope. In Rust, this concept is fully embraced, as variables not only hold data but also manage system resources like heap-allocated memory (`Box<T>`), file handles, and network sockets. The beauty of RAII in Rust lies in the fact that when a variable exits its scope, its destructor is automatically triggered, which ensures that any associated resources are properly cleaned up. This approach prevents memory leaks and guarantees safe and efficient resource management throughout the lifecycle of the variable. In Rust, the RAII pattern is implemented through the `Drop` trait, which defines the `drop(&mut self)` method that acts as a destructor for the object. The Rust compiler enforces the automatic invocation of this method in a deterministic manner. Specifically, the `drop` method is called when a variable leaves its syntactical scope, which happens when the variable is no longer needed. It is also triggered immediately before the variable is reassigned a new value. This ensures that resources are released properly, allowing objects to perform any necessary clean-up operations that are essential when following the RAII design pattern. The design of the `Drop` trait in Rust is crucial for maintaining memory safety without requiring manual intervention. Rust enforces a critical design rule in relation to the `Drop` and `Copy` traits: if a type implements

`Drop`, it cannot implement `Copy`, and vice versa. This design choice eliminates the possibility of implicitly copying objects that require explicit resource deallocation. By ensuring that a type cannot both copy and drop its resources, Rust prevents potential issues like double freeing or premature deallocation of memory. This mutual exclusion reinforces the safety guarantees that are central to Rust's memory model, ensuring that resources are managed correctly and preventing common bugs that could occur in systems programming. In addition to the automatic destruction mechanism provided by the `Drop` trait, Rust also provides a global function called `fn drop<T>(_x: T) {}`. This function allows for explicit control over the destruction of an object. By calling `drop`, the ownership of a value can be transferred to a new variable, forcing its destruction immediately, even before the end of its normal lifetime. This functionality can be useful in specific scenarios where more control over resource management is needed. Below is a graphical illustration of how the `Drop` trait functions in Rust:



```
struct Resource {
    name: String,
}
impl Drop for Resource {
    fn drop(&mut self) {
        println!("Releasing resource: {}", self.name);
  }
}
fn main() {
    let _res = Resource { name: String::from("RAII Example") };
    println!("Resource initialized.");
}
// The destructor is called automatically when _res goes out of scope
```

Figure 2.2: The `Drop` Trait Mechanism in Rust

RAII in Rust provides a reliable and deterministic way of managing system resources, which is one of the reasons why Rust is considered a safe and efficient systems programming language. By leveraging the `Drop` trait, Rust ensures that resources like memory, file handles, and network sockets are automatically and safely cleaned up when they are no longer needed. This prevents common pitfalls like memory leaks, double frees, and use-after-free errors. Rust's ownership and borrowing model, in combination with RAII, allows developers to write efficient, high-performance code while minimizing the risks of manual memory management. This powerful tool enables developers to create safe, fast, and reliable systems

without having to manage memory manually.

Traits

Polymorphism, refers to the ability of a single interface to support multiple underlying data types. In software engineering, polymorphism is a crucial principle that enables code reuse and abstraction, reducing redundancy according to the DRY (Don't Repeat Yourself) principle. Rust implements polymorphism primarily through traits, which define a collection of methods that a type can implement. In Rust, traits serve a role similar to interfaces in Java and C# or pure abstract classes in C++, allowing different types to share a common behavior while maintaining type safety. Unlike traditional inheritance-based polymorphism, Rust's trait system avoids the overhead of virtual function tables (VTABLEs) unless dynamic dispatch is explicitly used through **`&dyn TraitName`**. This enables more efficient, zero-cost abstractions at compile time through static dispatch. A trait defines a set of functions that types must implement. If a function does not take `self` as a parameter, it behaves as a **static method** rather than an instance method. Traits can also provide default method implementations, allowing types to override only specific behavior while inheriting a general implementation.

```
trait SomeTrait {
    fn some_operation(&self) -> i32;
}

struct MyType;
impl SomeTrait for MyType {
    fn some_operation(&self) -> i32 {
        42
    }
}
```

Figure 2.3: Traits in Rust

- **Trait Bounds:** Generic functions in Rust can enforce constraints on the types they accept using trait bounds (`T: SomeTrait`), ensuring compile-time

verification of method availability.

- **Supertraits:** A trait can require the implementation of another trait, forming a dependency hierarchy (`trait SubTrait: SuperTrait {}`).
- **Associated Types:** Traits can define **associated types** to specify generic parameters that vary between implementations.
- **Operator Overloading:** Traits such as `Add`, `Sub`, and `Mul` enable operator overloading.
- **Trait Objects:** Dynamic dispatch can be enabled using `dyn Trait`, incurring a runtime cost but allowing heterogeneous collections.

Rust's trait-based polymorphism provides a powerful yet efficient alternative to traditional OOP inheritance. By leveraging static dispatch and avoiding unnecessary memory overhead, Rust ensures high-performance, type-safe, and modular code design. The combination of trait bounds, default implementations, associated types, and dynamic dispatch makes Rust's polymorphism system both flexible and performant, aligning with modern software engineering principles.

Smart Pointers

In Rust, smart pointers play a crucial role in ensuring efficient and safe memory management by integrating seamlessly with the language's ownership model. Unlike raw pointers, they enforce strict rules on resource management, preventing common issues like memory leaks and data races at compile time. A fundamental example is `Box<T>`, which enables heap allocation while maintaining exclusive ownership of the stored value. This makes it particularly useful for recursive data structures that require dynamically sized storage. When multiple owners need access to the same value, `Rc<T>` provides reference counting in single-threaded contexts, while `Arc<T>` extends this capability to multi-threaded environments through atomic operations. To prevent memory from being retained unnecessarily due to cyclic references, `Weak<T>` offers non-owning pointers that do not contribute to reference counts. Rust also introduces mechanisms for controlled mutability. `RefCell<T>` allows modifying values even when they are borrowed immutably, enforcing borrowing rules dynamically at runtime instead of statically at compile time. `Cell<T>`, on the other hand, enables interior mutability by allowing value replacement without direct references. Another notable abstraction, `Cow<T>` (Clone-on-Write), optimizes memory usage by postponing data duplication until a modification is required. By leveraging these smart pointers, Rust provides both memory safety and flexibility, eliminating many of the pitfalls associated with manual memory management. The combination of strict ownership rules and specialized abstractions ensures that

memory-related errors are caught early, leading to more reliable and performant software.

2.3 The Role of Rust in Advancing Space Applications

On February 26, 2024, the U.S. White House issued a 19-page report [10] emphasizing the importance of adopting memory-safe programming languages in software development. The report specifically recommended transitioning away from languages such as C and C++ in favor of safer alternatives like Go, Java, Ruby, Swift, and Rust. This announcement, released through the Office of the National Cyber Director, was widely regarded as a signal of growing interest in Rust and other memory-safe languages. In this field, research on emerging technologies, including the use of Rust in space applications, has rapidly gained momentum. One notable example is the study under consideration [11], which explores Rust's integration into safety-critical space systems. The increasing complexity of space missions and the growing reliance on software-driven spacecraft systems highlight the critical importance of software safety and security. For instance, the proliferation of CubeSats (small satellites), widely deployed in Low Earth Orbit (LEO), reflect the growing shift toward miniaturization and increased accessibility in space technology. Traditionally, aerospace software has been developed in C due to its efficiency and established ecosystem. However, C's lack of inherent memory safety mechanisms leads to vulnerabilities such as buffer overflows, use-after-free errors, and null pointer dereferences. These weaknesses pose significant risks, including mission failure, hardware damage, and potential unauthorized access to spacecraft systems. Safety-critical software in space must meet strict reliability and robustness requirements. Standards for aerospace software and for automotive functional safety define methodologies to mitigate risk through rigorous testing, verification, and coding standards. However, many embedded and real-time operating systems used in spacecraft lack modern security mitigations such as Address Space Layout Randomization (ASLR) and non-executable stacks, making them susceptible to exploitation. Furthermore, traditional security practices in space applications have been inadequate, with many systems relying on security-by-obscurity rather than proactive defense mechanisms. A key strategy for enhancing software reliability is the selective replacement of C components with Rust, particularly in high-risk areas. By rewriting critical functionalities while maintaining compatibility through Foreign Function Interfaces (FFI), Rust can incrementally improve software security without requiring a complete overhaul of legacy systems. This approach has been demonstrated in satellite communication protocols, where Rust-based rewrites have identified and mitigated previously undetected vulnerabilities. In

this scenario, cross-compilation plays a crucial role in deploying Rust-based software to space-grade hardware. The Rust compiler supports multiple architectures, including ARM Cortex-M and PowerPC, both of which are commonly used in radiation-hardened spacecraft processors. By enabling Rust's safety features on these platforms, cross-compilation facilitates the integration of modern, memory-safe programming techniques into existing aerospace systems, enhancing their resilience against software faults and security threats.

2.3.1 The Embedded Rust Ecosystem

The applicability of a programming language in safety-critical space systems depends on its ability to efficiently manage system resources and interface with real-time hardware. Spacecraft, such as satellites, integrate multiple embedded systems that require robust software support for various microcontrollers and peripherals. To ensure hardware compatibility and maintainability, embedded software development often relies on Hardware Abstraction Layers (HALs), which allow device-independent driver implementation. Rust's `embedded-hal` ecosystem has grown significantly, now supporting a broad range of microcontrollers and peripherals, demonstrating its increasing maturity for space applications. The safety and security of embedded systems remain a critical concern, particularly in environments where software failures can have severe consequences. Rust's ecosystem includes security-focused tools such as flip-link, which prevents stack overflows in bare-metal environments by restructuring memory layout to ensure that overflows trigger hardware exceptions rather than silent data corruption. This proactive approach to memory safety is essential for space applications, where physical debugging is often impossible. Ensuring Rust's adoption in safety-critical systems requires compliance with industry standards. The High Assurance Rust initiative has introduced guidelines, establishing best practices for safe and secure embedded software development. Additionally, compiler qualification efforts have advanced significantly, with the Ferrocene Rust compiler recently achieving ISO 26262 (ASIL D) and IEC 61508 (SIL 4) certifications, demonstrating its reliability for high-assurance domains. Future efforts aim to extend these qualifications to aerospace standards such as DO-178C. These developments highlight the ongoing progress in adapting Rust for safety-critical embedded systems, reinforcing its potential as a viable alternative to traditional languages in the space sector.

2.3.2 Past Projects

- *Evaluation of RUST usage in space applications by developing BSP and RTOS targeting SAMV71*, initiated by ESA in 2023, aims to evaluate the viability of

the Rust programming language for space applications. The focus is on developing a lightweight Real-Time Operating System (RTOS) tailored for space missions, with a minimal feature set that supports the development of flight software. The project will include the creation of a Board Support Package (BSP) and a demonstration application for CubeSat-class missions, featuring basic functionalities such as UART communication, mode management, and sensor handling. The goal is to assess Rust's safety features and growing usage within the space sector, while aligning with ECSS software development practices. The project will also document feedback, issues encountered, and potential areas for improvement, providing valuable input for future space projects using Rust.

- The project *"cRustacea in Space – Co-operative RUST and C Embedded Applications in Space – Theory and Practice"*, conducted under ESA's Discovery & Preparation program, investigates the feasibility of using Rust as a programming language for onboard software development in space missions. Given the increasing complexity of space software requirements, traditional C-based approaches are being reassessed to enhance safety, reliability, and maintainability. The primary goal of this initiative is to assess Rust's suitability for real-time space applications while ensuring compliance with ESA's ECSS software engineering and product assurance standards. The study [12] concludes that **Rust is a viable alternative for space software development**, offering significant safety improvements over C. While Rust's learning curve remains a challenge, its structured memory management and modern programming paradigms make it a strong candidate for critical onboard applications.

2.3.3 What's Next

The future of Rust in space applications looks promising, but several key aspects need to be addressed to ensure its successful adoption. One of the primary challenges is aligning Rust with the strict requirements of ECSS (European Cooperation for Space Standardization) standards. Since space software demands high reliability and certification processes can be rigorous, defining clear pathways for Rust's qualification will be essential. Another crucial area for improvement is real-time performance. While Rust's memory safety and concurrency features offer clear advantages, further work is needed to guarantee deterministic execution and efficient task scheduling for hard real-time systems. This goes hand in hand with enhancing Rust's ecosystem for embedded systems and RTOS (real-time operating systems). Developing more specialized libraries and frameworks will be key to making Rust a viable alternative to traditional aerospace languages. In the upcoming sections, the simulation will not account for real-time operations, as the primary focus will be on the transmission of cryptographic keys and error correction techniques. One of

Rust's strongest assets, its memory safety model, could also be leveraged to improve fault tolerance in radiation-prone environments. Research into how Rust's type system and error-handling mechanisms can support software-based fault detection and mitigation strategies could provide significant benefits in spacecraft software development. Adoption within the space industry will also depend on fostering collaboration between space agencies, research institutions, and industry partners. Encouraging working groups dedicated to Rust in aerospace could accelerate progress, ensuring that common challenges are addressed collectively. At the same time, improving tooling for debugging, profiling, and formal verification will be necessary to gain confidence in Rust's reliability for mission-critical applications. To support long-term adoption, it will be important to establish standardized best practices for writing maintainable and efficient Rust code in space applications. Stability is another key consideration. Providing Long Term Support (LTS) versions of Rust would offer developers assurances regarding backward compatibility and the longevity of the software, both of which are crucial for space missions that span years or even decades. Comparative benchmarks with languages like C or Ada could help demonstrate Rust's strengths and identify areas for further optimization. Additionally, ensuring that Rust integrates smoothly with existing space software stacks would make its adoption more practical, allowing for a gradual transition rather than requiring a complete overhaul of existing systems. In conclusion, while Rust presents significant advantages for space software development, targeted efforts in these areas will be necessary to make it a standard choice for future space missions.

Chapter 3

Crystals Kyber Algorithm

3.1 Main Mathematical Results

This section aims to explore and provide an overview of the mathematical concepts underlying the Crystals Kyber algorithm, along with the existing security proofs.

3.1.1 Lattices and Fundamental Properties

In [13], a **lattice** L in $\mathbb{R}^n$ is defined as a discrete subgroup of $\mathbb{R}^n$. Formally, a lattice is defined as the integer span of a basis of $\mathbb{R}^n$:

$$L = \{a_1 v_1 + a_2 v_2 + \cdots + a_n v_n \mid a_i \in \mathbb{Z}\}, \tag{3.1}$$

where $\{v_1, \ldots, v_n\}$ is a basis of $\mathbb{R}^n$. The integer n is the **rank** or **dimension** of the lattice.

A lattice has multiple bases, but some are more suitable for computational applications than others. The quality of a basis plays a crucial role in many algorithms, such as lattice reduction techniques.

Given a basis $B = \{v_1, \ldots, v_n\}$ for a lattice L, the **fundamental domain** associated with B is defined as:

$$F(B) = \{t_1 v_1 + t_2 v_2 + \cdots + t_n v_n \mid 0 \leq t_i < 1\}. \tag{3.2}$$

The **determinant** (or volume) of a lattice is given by:

$$\det(L) = \mathrm{Vol}(F(B)) = |\det M(B)|, \tag{3.3}$$

where $M(B)$ is the matrix whose columns are the basis vectors.

This determinant is invariant under unimodular transformations, meaning that if B and B' are two different bases of L, then there exists a matrix $A \in SL_n(\mathbb{Z})$ such that:

$$M(B') = AM(B), \tag{3.4}$$

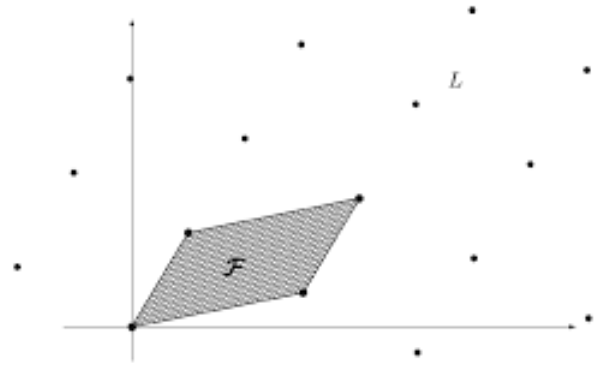


Figure 3.1: Fundamental domain of a lattice in $\mathbb{R}^2$.

ensuring that the volume remains unchanged:

$$\text{Vol}(F(B)) = \text{Vol}(F(B')). \tag{3.5}$$

The determinant provides a measure of the "density" of the lattice and is closely related to computational problems in lattice cryptography.

3.1.2 Lattice Problems

Lattice-based problems play a crucial role in modern cryptography, particularly in the development of quantum-resistant cryptographic schemes. Their underlying hardness assumptions serve as a robust foundation for secure encryption, digital signatures, and advanced cryptographic primitives. The NP-hardness of the problem underlying Crystals Kyber has already been established through reductions, leveraging some of the most extensively studied fundamental algorithms in lattice theory. Before listing some of the most important lattice problems, a clear definition of minimum distance in lattice theory is required. It is defined in [14] as follows:

Definition 1 *For any lattice L, the minimum distance of L is the smallest distance between any two lattice points:*

$$\lambda_1(L) = \inf\{\|x - y\| : x, y \in L, x \neq y\}.$$

The minimum distance of a lattice can also be defined as the length of the shortest nonzero vector in the lattice:

$$\lambda_1(L) = \inf\{\|v\| : v \in L, v \neq 0\}.$$

This follows from the fact that lattices are additive subgroups of $\mathbb{R}^n$, meaning they are closed under both addition and subtraction of their vectors.

3.1.3 Shortest Vector Problem

The exact version of the Shortest Vector Problem (SVP) has three primary formulations. Without loss of generality, we consider integer lattices with integral bases:

1. **Decision:** Given a lattice basis B and a real number $d > 0$, determine whether $\lambda_1(L(B)) \leq d$ or $\lambda_1(L(B)) > d$.
2. **Calculation:** Given a lattice basis B, compute the value of $\lambda_1(L(B))$.
3. **Search:** Given a lattice basis B, identify a nonzero vector $v \in L(B)$ such that $\|v\| = \lambda_1(L(B))$.

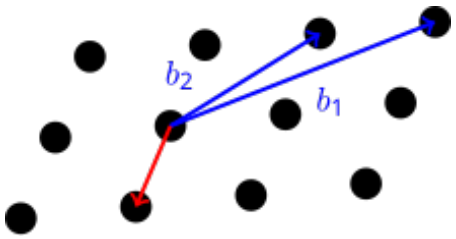


Figure 3.2: Shortest Vector Problem in a lattice L.

Clearly, solving the Calculation version directly provides a solution to the Decision problem. Formally, we express this as "Decision reduces to Calculation," denoted as Decision $\leq$ Calculation, highlighting the direction of reduction.

Conversely, the relationship Calculation $\leq$ Decision also holds. Using an oracle for the Decision problem, we can determine $\lambda_1(L(B))$ via binary search by systematically adjusting the parameter d. This condition holds because the minimum distance corresponds to the square root of an integer and is constrained between 1 and $n \det(B)^{1/n}$, as given by Minkowski's theorem. Since the determinant can be computed in polynomial time, this upper bound is also within $2^{\text{poly}(|B|)}$.

Additionally, it can be proved that the Search version of the problem is computationally equivalent to the other two formulations.

Approximate SVP

The **γ-approximate Shortest Vector Problem** (where $\gamma = \gamma(n) \geq 1$ is a function of the dimension n) has the following variations, again considering integer lattices:

1. **Decision (GapSVP$_\gamma$):** Given a lattice basis B and a positive integer d, decide whether $\lambda_1(L(B)) \leq d$ or $\lambda_1(L(B)) > \gamma d$.

2. **Estimation ($\mathbf{EstSVP}_\gamma$):** Given a lattice basis B, compute an approximation of $\lambda_1(L(B))$ within a factor of γ, meaning the output must satisfy $d \in [\lambda_1(L(B)), \gamma\lambda_1(L(B))]$.

3. **Search ($\mathbf{SVP}_\gamma$):** Given a lattice basis B, find a nonzero vector $v \in L(B)$ such that $0 < \|v\| \leq \gamma\lambda_1(L(B))$.

Setting $\gamma = 1$ recovers the exact versions of these problems. Moreover, as γ increases, the problems become computationally easier. More formally, the reductions hold as follows:

$$\mathrm{GapSVP}_\gamma \leq \mathrm{EstSVP}_\gamma \leq \mathrm{GapSVP}_{\gamma'}$$

for any $\gamma' \geq \gamma$, with an analogous relationship for SVP_γ.

Using a binary search strategy, we can establish that $\mathrm{GapSVP}_\gamma \leq \mathrm{EstSVP}_\gamma \leq \mathrm{GapSVP}_{\gamma'}$, indicating that these two variants are equivalent.

3.1.4 Closest Vector Problem

Definition 2 *Let $L \subset \mathbb{R}^n$ be a lattice. The **Closest Vector Problem (CVP)** consists of finding, for a given target vector $t \in \mathbb{R}^n$, a vector in L that is closest to t.*

In other words, the goal of CVP for a lattice L with target vector t is to find a vector $v_0 \in L$ such that:

$$\|v_0 - t\| = \min_{v \in L} \|v - t\|.$$

In its most general form, solving the exact versions of both the Shortest Vector Problem (SVP) and the Closest Vector Problem (CVP) is computationally challenging. However, in many practical applications, an approximate solution is sufficient. This leads to the following relaxed formulation.

Definition 3 *Let $L \subset \mathbb{R}^n$ be a lattice. The **Approximate Closest Vector Problem (apprCVP)** consists of finding, for a given target vector $t \in \mathbb{R}^n$, a lattice vector that is reasonably close to t.*

More precisely, for a given approximation factor $\gamma \geq 1$, the apprCVP requires finding a vector $v_0 \in L$ such that:

$$\|v_0 - t\| \leq \gamma \cdot \min_{v \in L} \|v - t\|.$$

3.1.5 Shortest Independent Vector Problem

The **Shortest Independent Vector Problem (SIVP)** is a fundamental computational challenge in lattice-based cryptography. The main objective is to minimize the length of the longest vector in a basis while preserving the structure of the lattice. In other words, the goal is to find a new basis for the same lattice where the longest vector is as short as possible.

Definition 4 *Given a basis $B = \{b_1, b_2, \ldots, b_n\}$ of a lattice L, find n linearly independent vectors $V = \{v_1, v_2, \ldots, v_n\}$ such that:*

$$\|v_i\| \leq \lambda_n, \quad \forall\, 1 \leq i \leq n.$$

A more general version of this problem is the **Approximate Shortest Independent Vector Problem (SIVP$_\gamma$)**, where an approximation factor $\gamma(n)$ is introduced.

Definition 5 *Given a basis $B = \{b_1, b_2, \ldots, b_n\}$ of a full-rank lattice L, output a set $V = \{v_i\} \subset L$ of n linearly independent lattice vectors such that:*

$$\|v_i\| \leq \gamma(n) \cdot \lambda(n)(L), \quad \forall\, i.$$

This problem has been extensively studied and is known to be **NP-hard**, as it can be reduced from the Closest Vector Problem (CVP).

3.1.6 Short Integer Problem

The **Short Integer Solution (SIS)** problem, first introduced in cryptography by Ajtai, is a fundamental computational problem in lattice-based cryptography. In [15], it is defined as the problem of finding a short integer vector in the kernel of a randomly chosen q-ary (i.e. mod q) matrix. Formally, it is defined as follows:

Definition 6 *Given positive integers n, m, q and a real parameter $\beta > 0$, the problem $SIS_{n,m,q,\beta}$ consists of the following search task:*

1. *Sample a uniformly random matrix $A \in \mathbb{Z}_q^{n \times m}$;*
2. *Given A, find a nonzero integer vector $z \in \mathbb{Z}^m$ such that:*

$$Az \equiv 0 \pmod{q}, \quad \textit{and} \quad \|z\|_2 \leq \beta.$$

The problem is trivially solvable using Gaussian elimination if no restriction is imposed on the norm of the solution. However, finding a short solution is computationally hard. For SIS to be intractable, it is necessary that $q > \beta$, since

otherwise, a trivial short solution such as $z = (q,0,\ldots,0)$ would always exist. A valid solution to SIS is not guaranteed for all parameter choices. However, it has been shown that a solution always exists when m and β are sufficiently large relative to n and q.

Ajtai introduced a family of *one-way* functions directly derived from the computational hardness of the Short Integer Solution (SIS) problem with appropriately chosen parameters.

Consider the function:

$$f : \mathbb{Z}_q^{n\times m} \times \mathbb{Z}_q^m \to \mathbb{Z}_q^{n\times m} \times \mathbb{Z}_q^n$$

defined as:

$$f(A, z) = (A, Az \mod q).$$

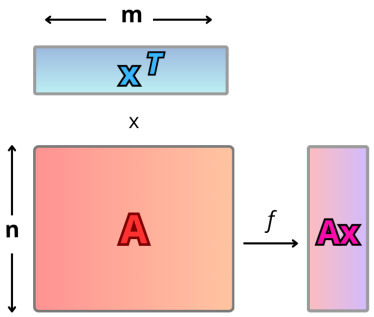


Figure 3.3: One-way function derived from SIS problem, introduced by Ajtai

It follows that, assuming the hardness of SIS, this family of functions is one-way, meaning that computing $f(A, z)$ is efficient, but inverting it (i.e., recovering z given only (A, Az)) is computationally infeasible. A one-way function of this type is a fundamental building block for various cryptographic constructions. In particular, its hardness assumption enables the design of several essential primitives, including pseudorandom generators, symmetric-key encryption schemes and digital signature schemes. These cryptographic mechanisms rely on the difficulty of inverting the function, ensuring security against adversarial attacks.

Beyond its one-wayness, the family of functions defined above, also presents collision resistance under the hardness assumption of the Short Integer Solution (SIS) problem. More precisely, given a randomly selected matrix A, it is computationally infeasible to find two distinct vectors $z, z' \in \{0,1\}^m$ such that:

$$Az \equiv Az' \pmod{q}.$$

This property further enhances its cryptographic utility, as collision-resistant function families serve as a foundation for the design of secure hash functions and other protocols.

3.1.7 Learning With Errors

The Learning With Errors (LWE) problem, initially introduced by Regev, requires to invert a (random) system of linear equations perturbed by short errors. [16] presents a more formal description of the simple version of the problem, upon which the majority of lattice cryptography relies.

Definition 7 *For positive integers m, n, q, and $\beta < q$, the $LWE_{n,m,q,\beta}$ problem asks to distinguish between the following two distributions:*

1. $(A, As + e)$, *where* $A \leftarrow \mathbb{Z}_q^{n \times m}$.
2. (A, u), *where* $A \leftarrow \mathbb{Z}_q^{n \times m}$, $s \leftarrow [\beta]^m$, $e \leftarrow [\beta]^n$ *and* $u \leftarrow \mathbb{Z}_q^n$.

The hardness of the problem $\mathrm{LWE}_{n,m,q,\beta}$, is based upon the presence of the additional "error" vector e, which removes the possibility of a Gaussian elimination attack. The parameters n, m, q, and β define the specific hardness of the problem. In its original formulation, the LWE problem employed an error distribution derived from a rounded Gaussian function. Specifically, a continuous, zero-centered Gaussian distribution with a given standard deviation was sampled, and the result was rounded to the nearest integer. This choice was crucial for the average-case to worst-case reduction proofs, which established that solving LWE is at least as difficult as solving certain worst-case lattice problems. However, subsequent research has demonstrated that this specific restriction is not strictly necessary, allowing alternative distributions to be used, such as the uniform distribution. In practice, CRYSTALS-Kyber utilizes a binomial distribution to generate error terms. This approach is often computationally more efficient, as it allows errors to be produced by summing a sequence of randomly generated bits rather than directly sampling from a uniform distribution over $[\beta]$. To account for different distributions that one could use, the LWE problem relative to the distribution of the secrets ψ can be defined as follows:

Definition 8 *For positive integers m, n, q, and a distribution ψ, the $LWE_{n,m,q,\psi}$ problem asks to distinguish between the following two distributions:*

1. $(A, As + e)$, *where* $A \leftarrow \mathbb{Z}_q^{n \times m}$, $s \leftarrow \psi^m$, $e \leftarrow \psi^n$.

2. (A, u)*, where* $A \leftarrow \mathbb{Z}_q^{n \times m}$ *and* $u \leftarrow \mathbb{Z}_q^n$*.*

In order to study the encryption scheme that stems from the formal description, some premises must be defined. For instance, the message μ, rather than being an arbitrary element in $\mathbb{Z}_q$, will now come from the set {0,1}. The key generation is modified to:

$$\text{sk} : s \leftarrow [\beta]^m, \tag{3.6}$$

$$\text{pk} : (A \leftarrow \mathbb{Z}_q^{m \times m}, t = As + e_1), \quad \text{where } e_1 \leftarrow [\beta]^m. \tag{3.7}$$

To encrypt a message $\mu \in \{0,1\}$, the encryptor chooses $r, e_2 \leftarrow [\beta]^m$ and $e_3 \leftarrow [\beta]$, and outputs:

$$u^T = r^T A + e_2^T, \tag{3.8}$$

$$v = r^T t + e_3 + \frac{q}{2}\mu. \tag{3.9}$$

To decrypt, one computes:

$$v - u^T s = r^T(As + e_1) + e_3 + \frac{q}{2}\mu - r^T As + e_2^T s. \tag{3.10}$$

Since the $r^T As$ terms cancel out, the output is:

$$e + \frac{q}{2}\mu, \tag{3.11}$$

where $e \in [2m\beta^2 + \beta]$, and so if the parameters are set such that $2m\beta^2 + \beta < q/4$, the decryptor is able to obtain μ by checking whether the value is closer to 0 or $q/2$.

The value $\frac{q}{2}$ is used as an encoding for the bit μ: since the message μ is binary (i.e., $\mu \in \{0,1\}$), it must be mapped into a larger numerical space, namely the field $\mathbb{Z}_q$. The encoding scheme is designed such that:

- If $\mu = 0$, the ciphertext value v should be close to 0.
- If $\mu = 1$, the ciphertext value v should be far from 0 but not arbitrarily large.

A natural choice is to place the central value of the ring $\mathbb{Z}_q$ as the representation of 1. In a cyclic ring modulo q, the midpoint is given by $\frac{q}{2}$. Upon decryption, the receiver obtains (3.11) where e is the total noise term. To ensure correct decryption, the noise e must remain sufficiently small relative to $\frac{q}{2}$:

- If the result is close to 0, the receiver decodes $\mu = 0$.

- If the result is close to $\frac{q}{2}$, the receiver decodes $\mu = 1$.

This encoding guarantees that the two possible values of μ are well separated, making decoding more robust against small noise perturbations. The error e must satisfy the condition:

$$|e| < \frac{q}{4},$$

ensuring that the received value remains clearly distinguishable between 0 and $\frac{q}{2}$. If the noise were larger, it could shift $v - u^T s$ toward the incorrect threshold, leading to a decryption error.

Using $\frac{q}{2}$ to represent 1 is a strategic choice that:

- Clearly differentiates the two binary values in $\mathbb{Z}_q$.
- Minimizes decryption errors caused by noise.
- Allows for correct decoding using a simple thresholding mechanism.

3.2 Hardness Proof By Reduction

3.2.1 Computational Complexity

Computational complexity theory categorizes problems based on their computational difficulty. The primary complexity classes include:

P Problems

Problems in class P can be solved efficiently in polynomial time using a deterministic Turing machine. These problems have a time complexity of the form:

$$T(n) = O(n^k), \quad \text{for some constant } k.$$

Examples include:

- Basic arithmetic operations (addition, multiplication, etc.).
- Sorting algorithms.
- Shortest path algorithms (e.g., Dijkstra's algorithm).

NP (Non-deterministic Polynomial-Time Problems)

Class NP consists of problems where a given solution can be **verified** in polynomial time, though finding the solution may take exponential time. Their time complexity is generally of the form:

$$T(n) = O(2^n) \text{ or higher.}$$

NP-Complete Problems

NP-Complete problems belong to NP and are the hardest in this class. They have the property that if any NP-Complete problem is solved in polynomial time, then all NP problems can be solved in polynomial time ($P = NP$). These problems also satisfy polynomial-time reducibility.

Examples:

- Travelling Salesman Problem.
- Knapsack Problem.
- Graph Coloring.

NP-Hard Problems

NP-Hard problems are at least as hard as NP problems but are not necessarily in NP. This means they may not have solutions verifiable in polynomial time.

Examples:

- Halting Problem.
- Certain optimization problems.

The P vs. NP Problem

A fundamental open question in computer science is whether:

$$P = NP.$$

If true, this would imply that all NP problems can be solved in polynomial time, which would have profound implications for cryptography, optimization, and artificial intelligence.

- **P problems** are efficiently solvable.
- **NP problems** are difficult to solve but easy to verify.

- **NP-Complete problems** are among the hardest in NP and are interconvertible.
- **NP-Hard problems** are even more complex and may not be verifiable in polynomial time.

3.2.2 LWE Hardness By Reduction

There have been numerous and diverse studies and attempts to analyze these problems from various perspectives. However, the paper in question [17] provides a thorough examination of the obtained results and the reasoning process that leads to considering the hardness of Learning With Errors (LWE) problems sufficiently strong for its use in cryptography. Cryptographic security often relies on computational problems that belong to the category of *average-case problems*, such as the Short Integer Solution (SIS) and LWE problems. In these cases, the adversary is provided with a randomly generated instance of the problem. For instance, in SIS, the adversary receives a uniformly random matrix $A \in \mathbb{Z}_q^{n \times m}$. Successfully breaking SIS requires designing an algorithm capable of finding a short, nonzero vector z in the kernel of A, but only for a small fraction of possible matrices A.

Conversely, computational complexity theory is primarily concerned with *worst-case problems*, where an algorithm must reliably solve *every* instance of the problem with a high probability of success. Notable examples include the approximate Closest Vector Problem (γ-CVP) and the approximate Shortest Vector Problem (γ-SVP). The theoretical foundations of worst-case problems are well established, making their complexity easier to analyze compared to average-case problems.

Reduction from Worst-Case to Average-Case Problems

A common strategy for analyzing the complexity of an *average-case problem* is to establish an *efficient reduction* from a well-studied *worst-case problem*. This reduction ensures that each instance of the worst-case problem can be transformed into an instance of the average-case problem. If such a reduction exists, then solving the average-case problem is at least as difficult as solving the worst-case problem. Consequently, any known hardness results for the worst-case problem also apply to the average-case setting. One of the key results was established by Peter Van Emde Boas in 1981, demonstrating that CVP is NP-hard by leveraging the well-known Subset Sum problem as the basis for the proof. Over time, several results have shown that SVP can be reduced to CVP, formally SVP $\leq$ CVP, implying that SVP is at least as difficult as CVP. Moreover, one of the longstanding open questions in cryptography was whether the security of cryptographic protocols could be rigorously based on the worst-case hardness of a well-characterized computational problem. This question was ultimately addressed by Ajtai, who demonstrated

that one-way functions could be constructed based on the worst-case hardness of the Short Integer Solution (SIS) problem. Furthermore, Ajtai established a key theoretical result, showing that the average-case hardness of SIS is tightly connected to the worst-case hardness of the approximate Shortest Vector Problem (γ-SVP).

Hardness of SIS

Theorem 1 (Ajtai, informal) *For any $m = poly(n)$, any $\beta > 0$, and any sufficiently large $q \geq \beta \cdot poly(n)$, solving the $SIS_{n,m,q,\beta}$ problem is at least as hard as solving γ-SVP on worst-case n-dimensional lattices with high probability, for some approximation factor $\gamma = \beta \cdot poly(n)$.*

Hardness of Decision-LWE

Similarly, Regev established a corresponding reduction for the Decision-LWE problem, demonstrating its worst-case hardness by linking it to γ-SVP in a quantum computational setting. This result was later extended to the classical setting by Peikert, leading to the following theorem:

Theorem 2 (Regev, Peikert, informal) *For any $m = poly(n)$, any $q \leq 2^{poly(n)}$, and any discretized Gaussian distribution χ with variance $\alpha q \geq 2\sqrt{n}$, solving the Decision-$LWE_{n,m,q,\chi}$ problem is at least as hard as solving γ-SVP on worst-case n-dimensional lattices with high probability, for some approximation factor $\gamma \approx n/\alpha$.*

In conclusion, the established reductions between SIS, LWE, and γ-SVP highlight the importance of understanding the worst-case hardness of γ-SVP.

3.3 NIST Submission

Kyber is an IND-CCA2 secure key encapsulation mechanism (KEM), described in detail in the official documentation [18]. In this project, round 3 version of Kyber has been taken as the reference implementation, following the official documentation guidelines. Its security relies on the difficulty of solving the module learning-with-errors (MLWE) problem. The construction of Kyber proceeds in two stages: first, an IND-CPA-secure public-key encryption scheme, known as Kyber.CPAPKE, is introduced to encrypt fixed-length messages of 32 bytes; then, a slightly modified Fujisaki–Okamoto (FO) transform is applied to convert this encryption scheme into an IND-CCA2-secure KEM, referred to as Kyber.CCAKEM when emphasizing its resistance to chosen-ciphertext attacks.

3.3.1 Kyber.CPAPKE: Encryption Scheme

Kyber.CPAPKE is an encryption scheme derived from the LPR approach originally proposed by Lyubashevsky, Peikert, and Regev for Ring-LWE, with its conceptual origins extending back to Regev's first LWE-based scheme and even to the NTRU cryptosystem by Hoffstein, Pipher, and Silverman. In Kyber, the principal modification involves replacing Ring-LWE with Module-LWE, thereby enhancing both flexibility and security. Furthermore, the method for generating the public matrix A follows the strategy outlined by Alkım, Ducas, Pöppelmann, and Schwabe, while public keys and ciphertexts are compressed via bit-dropping techniques based on learning-with-rounding. The scheme is parameterized by a set of integers, with $n = 256$ and $q = 7681$ fixed throughout, and parameters d_u, d_v, and d_t set to 11, 3, and 11 respectively; the parameters k and η are varied to achieve different security levels. This structure underlines Kyber's commitment to efficiency and robustness in lattice-based encryption. The Kyber.CPAPKE encryption scheme follows these steps:

Key Generation

1. Sample $\mathbf{s}, \mathbf{e} \leftarrow \chi$.

2. Compute:

$$\text{sk} = \mathbf{s}, \quad \text{pk} = \mathbf{t} = A\mathbf{s} + \mathbf{e}$$

Encryption

1. Sample $\mathbf{r}, \mathbf{e_1}, \mathbf{e_2} \leftarrow \chi$.

2. Compute:

$$\mathbf{u} \leftarrow A^T\mathbf{r} + \mathbf{e_1}$$

$$v \leftarrow \mathbf{t}^T\mathbf{r} + \mathbf{e_2} + \text{Enc}(m)$$

3. The ciphertext is:

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

Decryption

1. Compute:

$$m = \text{Dec}(v - \mathbf{s}^T\mathbf{u})$$

3.3.2 Design Decisions

The design of Kyber is primarily based on the module variant of the Ring-LWE encryption scheme introduced by Lyubashevsky, Peikert, and Regev (LPR). This scheme integrates a bit-dropping technique for efficiency and incorporates advancements from previous lattice-based encryption implementations such as NewHope.

A key difference between Kyber and conventional Ring-LWE schemes lies in the utilization of module lattices. While previous schemes performed operations of the form $As + e$, where all variables were polynomials in a structured ring, Kyber instead treats A as a small matrix (e.g., 3×3) over a polynomial ring of fixed size, and s, e as vectors over the same ring.

Module-LWE: Balancing Structure and Efficiency

Kyber employs Module-LWE instead of either Ring-LWE or standard LWE. The trade-offs are as follows:

- **Ring-LWE** provides high efficiency in terms of speed and memory but introduces algebraic structures that may allow more efficient attacks.
- **Standard LWE** eliminates such structure, improving security, but at the cost of significantly increased computational overhead.
- **Module-LWE** balances these factors, maintaining efficiency while reducing the algebraic structure that could be exploited.

The used ring structure is:

$$R = \mathbb{Z}_q[X]/(X^{256} + 1), \quad q = 7681$$

In Kyber's case, the chosen parameters reduce structure compared to Ring-LWE while allowing better scalability and performance when encrypting fixed-size messages (e.g., 256-bit messages), using more generic rings.

Active Security Considerations

Unlike earlier passively secure KEMs used in transitional post-quantum security settings (e.g., TLS migrations), Kyber is defined as an IND-CCA2 secure KEM. Although passively secure KEMs have advantages such as higher tolerance for failure probability and faster decapsulation (since they do not require a CCA transform), active security is necessary for many cryptographic applications, including:

- Public-key encryption (via KEM-DEM constructions).
- Authenticated key exchange.

- Secure ephemeral key caching in protocols like TLS.

Additionally, Kyber's CCA transform protects against implementation vulnerabilities. For instance, passive schemes may fail to detect incorrect noise values (e.g., all-zero noise). Kyber's re-encryption step ensures such anomalies are immediately detected.

Role of the Number Theoretic Transform (NTT)

The Number Theoretic Transform (NTT) is a fundamental component in lattice-based cryptography due to its ability to significantly accelerate polynomial multiplications while maintaining a compact memory footprint. Unlike traditional multiplication methods, NTT-based multiplication does not introduce additional memory overhead. Consequently, it has become standard practice to select cryptographic parameters that optimize the efficiency of NTT operations. Several post-quantum cryptographic schemes, including NewHope, integrate NTT operations directly into their design. Specifically, Kyber defines the matrix A in its public-key encryption scheme (Kyber.CPAPKE) directly in the NTT domain. This decision necessitates that all multiplications involving A must also occur in the NTT domain. The selective integration of NTT in Kyber balances computational efficiency and data compression. By defining matrix A in the NTT domain and ensuring that multiplications are performed consistently in this representation, Kyber minimizes unnecessary computations while preserving security and compact ciphertext sizes.

Uniform Generation of Matrix A

The approach adopted for generating the public uniformly random matrix A follows the "against-all-authority" principle. In this model, the matrix A is not a fixed system parameter but is instead freshly generated as part of each public key. This strategy offers two main advantages: first, it eliminates the need to discuss the specifics of how a uniformly random system parameter is generated, thus avoiding potential disagreements over this process. Second, it safeguards against the "all-for-the-price-of-one" attack scenario, where an attacker could dedicate considerable computational resources to finding a short basis for the lattice spanned by A and subsequently use this basis to compromise all users. The trade-off associated with this approach is the expansion of the matrix A, from a random seed during key generation and encapsulation.

Noise Distribution: Binomial vs. Gaussian

In earlier implementations, discrete Gaussian noise was commonly used. However, this approach was found to be inefficient and vulnerable to timing attacks.

The effectiveness of attacks against LWE encryption primarily depends on the standard deviation and entropy of the noise, rather than its precise distribution. Consequently, more efficient and secure noise distributions, such as the centered binomial distribution, have gained attention. For instance, the Kyber encryption scheme adopts centered binomial noise, leveraging LWE rather than learning-with-rounding (LWR) as the underlying problem. Furthermore, Kyber's Compress function introduces additional noise during ciphertext compression, which enhances security.

Handling Decapsulation Failures

A key design decision in Kyber revolves around whether to permit decapsulation failures. Although setting the failure probability to zero simplifies security proofs and mitigates concerns about failure-based attacks, this decision comes with trade-offs:

- Decreasing the noise reduces security against lattice-based attacks.
- Increasing the lattice dimension to counteract this loss in security leads to reduced efficiency.

Kyber opts for a failure probability of less than 2^{-140}, ensuring that failures are negligible while striking an optimal balance between security and performance.

3.3.3 Kyber's Fujisaki-Okamoto Transform

Kyber is the selected Key Encapsulation Mechanism (KEM) from the NIST Post-Quantum Cryptography Standardization project and is set to become the standard post-quantum KEM. Similar to most post-quantum KEMs, Kyber is built upon a CPA-secure Public Key Encryption (PKE) scheme, which is then transformed into an IND-CCA-secure KEM using the Fujisaki-Okamoto (FO) transform. However, Kyber does not employ the standard FO transform but rather a modified version, leading to different security implications.

The modified FO transform used in Kyber, referred to as $\mathtt{FO}_{\mathtt{Kyber}}$, deviates from the standard $\mathtt{FO}_{\perp}$ construction in two key areas:

- The randomness used in encryption is derived not only from the message but also from an additional hash of the public key.
- The key derivation process differs by replacing the message with an intermediate pre-key and utilizing a hash of the ciphertext instead of the ciphertext itself.

These modifications complicate existing security proofs, particularly those leveraging implicit rejection techniques in the quantum random oracle model (QROM). Previous direct proof attempts either relied on explicit rejection or introduced additional complexity, such as collision terms that impact the security bound. This work explores an alternative proof strategy that remains closer to prior methods while introducing a slightly looser bound.

Comparison: Standard FO vs. $\mathbf{FO_{Kyber}}$

Process	**Standard FO**	**$\mathbf{FO_{Kyber}}$**
Randomness Generation	$r = G(m)$	$(\tilde{K}, r) = H_1(m, H_3(pk))$
Key Derivation	$K = H(m, c)$	$K = H(\tilde{K}, H_2(c))$
Ciphertext Validation	Re-encrypt and compare with c	Compare $H_2(c)$ instead
Failure Handling	PRF with ciphertext input	PRF with hashed ciphertext input

Table 3.1: Differences between Standard FO and $\mathrm{FO}_{\mathrm{Kyber}}$.

The security proof of FO-based KEMs often relies on an implicit rejection technique, where decryption queries are simulated by manipulating the random oracle used in key derivation. However, the modifications in $\texttt{FO}_{\texttt{Kyber}}$ introduce the following obstacles:

- The introduction of $\tilde{K}$ instead of m breaks existing proof techniques that rely on plaintext knowledge.
- The use of $H_2(c)$ instead of the ciphertext c itself prevents direct message-ciphertext validation.
- The collision resistance of H_2 becomes a crucial factor, leading to an additional collision term in the security bound.

Due to these issues, previous proof strategies either fail entirely or require additional assumptions. The final security bound for $\texttt{FO}_{\texttt{Kyber}}$ now includes:

- A collision term associated with H_2.
- A term accounting for the indistinguishability of $\tilde{K}$.

- A slightly looser bound compared to the standard FO transform.

Ultimately, these complications suggest that reverting to the standard FO transform would provide a more straightforward and provable security guarantee, avoiding the additional complexity introduced by $\texttt{FO}_{\texttt{Kyber}}$.

3.4 Security Analysis

Reporting from the official supporting documentation of NIST submission [18], an evaluation of Kyber's security is provided, under various assumptions. The underlying hard problem that ensures the security of the schemes is the Module-LWE problem. This problem involves distinguishing between uniform samples $(a_i, b_i) \in \mathbb{R}_q^k \times \mathbb{R}_q$ and samples $(a_i, b_i) \in \mathbb{R}_q^k \times \mathbb{R}_q$ where $a_i \in \mathbb{R}_q^k$ is uniformly distributed, and $b_i = a_i^T s + e_i$, with $s \in \mathbb{B}^k$ being the same for all samples, and $e_i \in \mathbb{B}$ being freshly chosen for each sample. More formally, for an algorithm A, this can be defined as:

$$\begin{aligned}\mathrm{Adv}_{\mathrm{mlwe}}^{m;k;\eta}(A) = \Big|&\Pr[b' = 1 : A\,(R_q^{m\times k}; (s,e) \in \mathbb{B}^k \times \mathbb{B}^m; b = A^T s + e)] \\ &- \Pr[b' = 1 : A\,(R_q^{m\times k}; b \in \mathbb{R}_q)]\Big| \end{aligned} \tag{3.12}$$

One of the most important results is:

Theorem 3 *Suppose XOF and G are random oracles. For any adversary A, there exist adversaries B and C with roughly the same running time as that of A such that*

$$Adv_{cpa}^{Kyber:CPAPKE}(A) \leq 2 \cdot Adv_{mlwe}^{k+1;k;\eta}(B) + Adv_{prf}^{PRF}(C).$$

The proof of this theorem is easily obtained by noting that, under the MLWE assumption, public-key and ciphertext are pseudo-random.

Kyber.CCAKEM is obtained via a slightly tweaked Fujisaki-Okamoto transform applied to Kyber.CPAPKE. The following concrete security statement proves Kyber:CCAKEM's IND-CCA2-security when the hash functions G and H are modeled as random oracles.

Non-tight Reduction from MLWE in the QROM

In the quantum random oracle model (QROM), it has been demonstrated that Kyber.CCAKEM is IND-CCA2 secure, assuming that Kyber.CPAPKE is IND-CPA secure. A tighter reduction can be achieved by assuming that the base scheme Kyber.CPAPKE is pseudo-random. Pseudo-randomness requires that, for any message m, the ciphertext $(c_1, c_2) = \text{Kyber.CPAPKE.Enc}(pk; m)$ is computationally

indistinguishable from a random ciphertext $(\text{Compress}_q(u; du), \text{Compress}_q(v; dv))$, where (u, v) are uniformly distributed. The proof of Kyber.CPAPKE's IND-CPA security confirms that it is tightly pseudo-random under the Module-LWE hardness assumption.

Theorem 4 *Suppose XOF, H, and G are random oracles. For any quantum adversary A that makes at most q_{RO} many queries to quantum random oracles XOF, H, and G, there exist quantum adversaries B and C of roughly the same running time as that of A such that*

$$Adv_{cca}^{Kyber.CCAKEM}(A) \leq 4q_{RO} \cdot q \cdot Adv_{mlwe}^{k+1;k;\eta}(B) + Adv_{prf}^{PRF}(C) + 8q_{RO}^2\epsilon.$$

In the following table, classical and quantum core-SVP hardness of the different proposed parameter sets of Kyber are listed, together with the claimed security level. Complexities are given in terms of the base-2 logarithm of the number of operations.

core-SVP (classical)	core-SVP (quantum)	Claimed security level	
Kyber512	112	102	1 (AES-128)
Kyber768	178	161	3 (AES-192)
Kyber1024	241	218	5 (AES-256)

Table 3.2: Kyber parameter sets and corresponding security level.

However, the above security bound is non-tight and therefore can only serve as an asymptotic indication of Kyber.CCAKEM's CCA-security in the quantum random oracle model.

Tight Reduction Under Non-Standard Assumption.

A tight security bound in the Quantum Random Oracle Model (QROM) can be established by assuming that a deterministic variant of Kyber.CPAPKE, referred to as DKyber.CPAPKE, is pseudo-random in the QROM. In this version, the random values r used during encryption are deterministically generated from the message m, i.e., $r := G(m)$. The pseudo-randomness property for DKyber.CPAPKE ensures that the encryption (c_1, c_2) of a randomly chosen message is computationally indistinguishable from a random ciphertext of the form $(\text{Compress}_q(u; du), \text{Compress}_q(v; dv))$, where (u, v) are uniformly distributed. While in the classical Random Oracle Model (ROM), the pseudo-randomness of DKyber.CPAPKE is tightly equivalent to the Module-LWE (MLWE) problem, the reduction in the QROM is non-tight. This non-tightness is reflected in the term $q_{RO} \cdot q \cdot \text{Adv}_{\text{mlwe}}^{k+1;k;\eta}(B)$ in Theorem 3, which leads to the following security bound:

$$\begin{aligned}\mathrm{Adv}_{\mathrm{cca}}^{\mathrm{Kyber.CCAKEM}}(A) \leq\ & 2 \cdot \mathrm{Adv}_{\mathrm{mlwe}}^{k+1;k;\eta}(B) \\ & + \mathrm{Adv}_{\mathrm{pr}}^{\mathrm{DKyber.CPAPKE}}(C) \\ & + \mathrm{Adv}_{\mathrm{prf}}^{\mathrm{PRF}}(D) + 8q_{RO}^2\epsilon. \end{aligned} \tag{3.13}$$

It should be noted that no quantum attack on deterministic Kyber.CPAPKE has been identified that performs better than solving the MLWE problem.

3.4.1 Attacks on the Underlying MLWE Problem

In the realm of post-quantum cryptography, the Module Learning With Errors (MLWE) problem, when applied to systems such as the Kyber cryptosystem, is typically modeled as an extension of the Learning With Errors (LWE) problem. This section explores several methods utilized to attack the MLWE problem, with a particular focus on the Block Korkine-Zolotarev (BKZ) algorithm.

3.4.2 The SVP Oracle

The **SVP oracle** is a theoretical tool designed to solve the Shortest Vector Problem in lattices. The SVP problem involves finding the shortest non-zero vector in a lattice, and the oracle provides an idealized solution by returning this shortest vector immediately. In cryptanalysis, the SVP oracle plays a crucial role, as algorithms such as BKZ rely on it to evaluate lattice reduction performance. By offering a "perfect" solution, it serves as a reference point for analyzing the efficiency of lattice-based cryptographic attacks.

3.4.3 The BKZ Algorithm

The **BKZ algorithm** (Block Korkine-Zolotarev) is a lattice reduction method that operates by splitting the lattice into smaller blocks of vectors. Each block is processed using an SVP oracle. The frequency with which the SVP oracle is queried depends on the lattice dimension and the size of the blocks involved. BKZ's main objective is to find short vectors within a lattice, which can then be leveraged to break cryptosystems like Kyber. The computational complexity of the BKZ algorithm is influenced by the block size and the dimension of the lattice.

3.4.4 Solving the SVP Oracle: Enumeration and Sieving

The two primary techniques used to solve the SVP oracle within the BKZ framework are **enumeration** and **sieving**. These techniques offer different ways to efficiently

identify short vectors in lattices, as an approximation of the ideal performance of the oracle. In BKZ, two primary methods are employed to solve the SVP oracle: enumeration and sieving, each offering distinct performance characteristics. Enumeration algorithms are known for their super-exponential time complexity, while sieving algorithms exhibit exponential time complexity. Empirical results suggest that enumeration is more efficient for lattices of smaller dimensions. However, sieving algorithms are expected to surpass enumeration in higher dimensions, making sieving a more practical choice for large-scale lattice-based cryptanalysis, such as attacks against Kyber. Recent advancements in sieving techniques have minimized the performance gap between enumeration and sieving, especially for exact-SVP problems in dimensions ranging from 60 to 80. However, sieving algorithms are considerably more memory-intensive, with their time and memory complexities growing exponentially. This leads to practical challenges as memory usage increases beyond the capacity of fast local memory (RAM). To estimate the practical performance of both techniques, a conservative lower bound for their efficiency is calculated within the RAM model, assuming that memory access is free of cost. Under this assumption, sieving outperforms enumeration for dimensions above 250. For Kyber, where dimensions as low as 390 are of interest, this RAM model provides a conservative lower bound for the performance of both enumeration and sieving. Recent improvements in sieving algorithms, such as the application of Locality Sensitive Hashing (LSH), have enhanced their efficiency by reducing the hidden sub-exponential factor. Additionally, quantum algorithms, like Grover's search, further lower the complexity of sieving techniques. The classical cost estimate for both primal and dual attacks is $20.292b$, while the quantum cost estimate is $20.265b$ where b is the lattice dimension in bits.

3.4.5 Attacks against MLWE

The main attacks on the MLWE problem in the Kyber cryptosystem rely on the BKZ algorithm, which can be used in two distinct attack strategies: primal and dual.

Primal Attack

The **primal attack** involves creating a unique-SVP instance derived from the LWE problem and then applying the BKZ algorithm to find a short vector in the lattice. The success of this attack hinges on the ability to find a vector whose norm satisfies specific conditions, allowing the cryptosystem to be compromised.

Dual Attack

The **dual attack** targets the dual lattice by searching for a short vector within it. The difficulty of this attack is influenced by the length of the vector found, and its effectiveness depends on distinguishing LWE samples from a uniform distribution.

3.4.6 Algebraic Attacks on Kyber

At present, the predominant attacks on the MLWE instance underlying Kyber are lattice-based, particularly focusing on algorithms like BKZ. However, algebraic attacks that exploit the structure of ideal lattices have been considered in theoretical discussions. While these algebraic methods are not yet a significant practical threat to the security of Kyber, there is a possibility that future developments in quantum algorithms could target ideal-SVP problems. Should such attacks be realized, they could pose a potential risk to lattice-based cryptosystems such as Kyber in the future.

Chapter 4

Error Detection and Correction

Turbo codes have gained significant attention due to their exceptional error correction capabilities, achieving performance close to the Shannon limit. Notably, they can attain very low bit error rates even at low signal-to-noise ratio (SNR) conditions.

4.0.1 Performance Comparison

Turbo codes, introduced by Berrou et al. in 1993 [19], represent a significant breakthrough in coding theory. These codes use convolutional codes, pseudo-random interleaving, and maximum a posteriori probability MAP iterative decoding to achieve error correction performance close to Shannon's capacity limit. Specifically, turbo codes achieve a bit error rate (BER) that is only 0.7 dB below the channel capacity limit in an additive white Gaussian noise AWGN channel. Remarkably, this performance is achieved with lower computational complexity compared to traditional convolutional codes. As clearly shown in [20], the ability of turbo codes to approach the theoretical limits of error correction makes them highly applicable to satellite communication systems, where low error rates are critical. In satellite communications, turbo codes are particularly advantageous because they can operate efficiently in the challenging AWGN environment, commonly encountered in deep-space communication scenarios. The ability to reduce code rates further enhances communication reliability by minimizing Bit Error Rates (BER). By exploiting the channel's bandwidth efficiently, turbo codes can overcome power limitations in satellite systems, enabling better utilization of available resources. A BER below 10^{-5} is generally considered to be acceptable in many engineering applications. In addition, since the performance of each coding mode has a large

difference, the Signal to Noise Ratio SNR scale in the simulation is non-uniform to make the results more obvious. In Figure 4.1, it has been shown that turbo codes

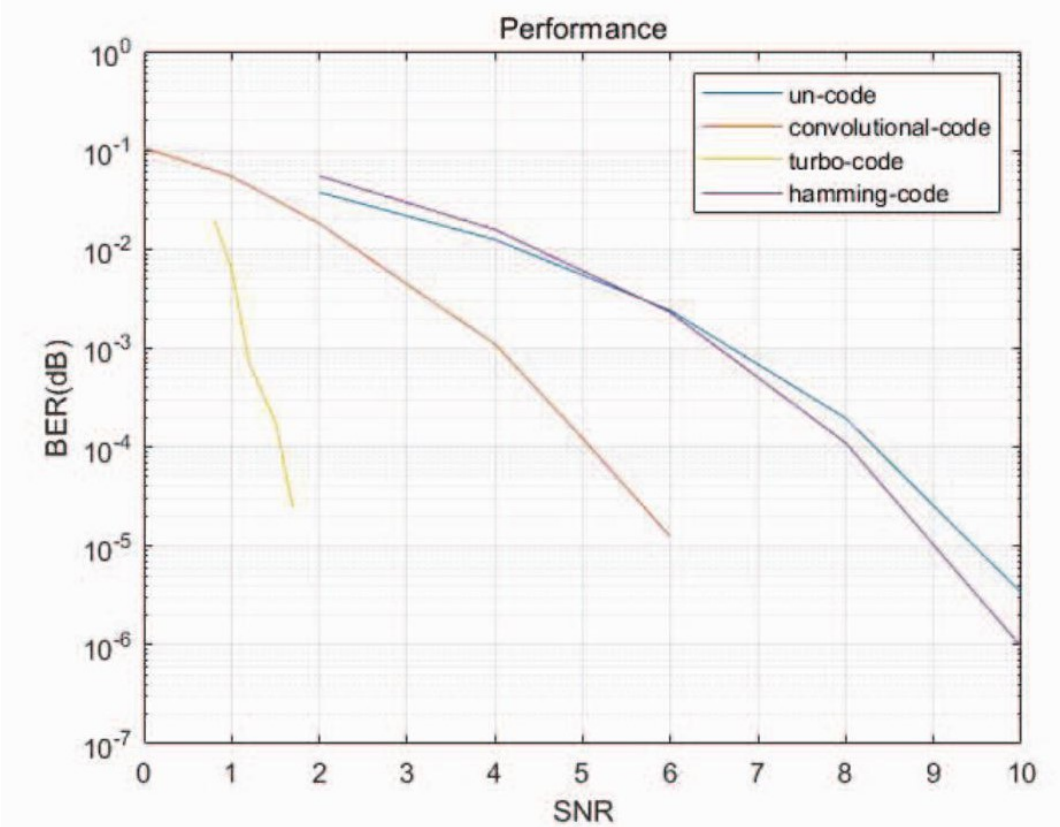


Figure 4.1: Performance comparison of different encoding methods

have an obvious advantage over convolutional codes and Hamming codes, which can achieve a very low BER with a small SNR.

Turbo Codes Performance Analysis

The Turbo code algorithm will be thoroughly analyzed in the following chapters; however, its expected behavior is preliminarily examined in this section to evaluate its suitability as a viable solution within this context, as shown in [21]. 4.3 illustrates the performance of turbo codes as a function of the number of decoder iterations, with uncoded BER provided for comparison. After the first iteration, the turbo decoder's performance aligns closely with that of convolutional codes. Increasing the number of iterations enhances decoding performance: for instance, a gain of approximately 1.2 dB is observed between the first and second iterations at a BER of 10^{-4}. This improvement continues up to the eighth iteration, beyond which the

performance gain diminishes significantly (e.g., only 0.1 dB improvement between the eighth and fourteenth iterations at BER 10^{-4}). However, increasing the number of iterations also raises the computational complexity of the decoding process.

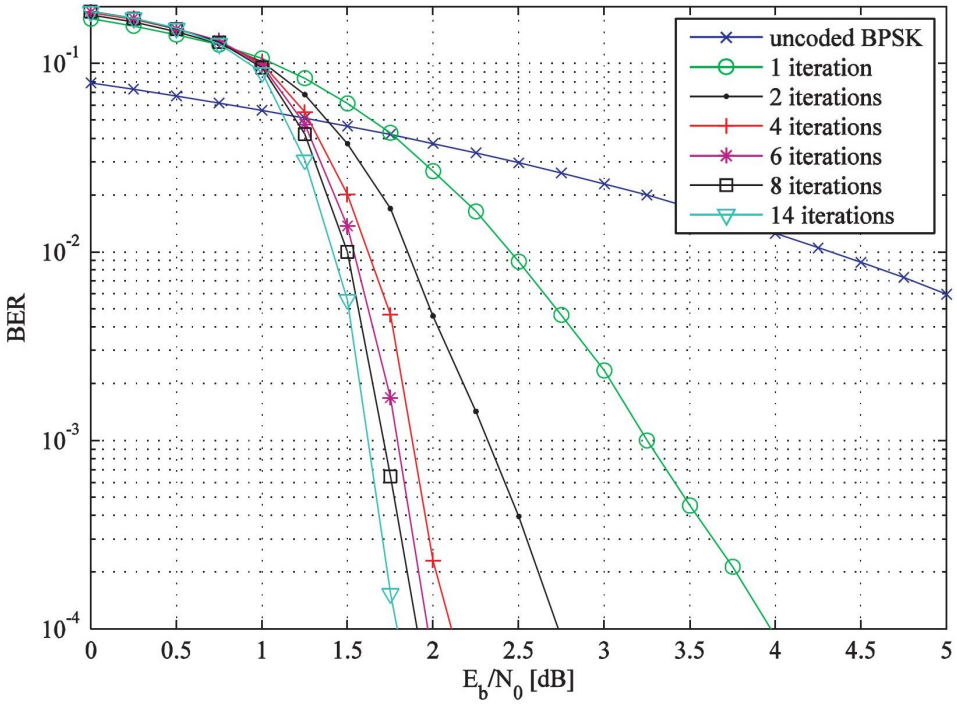


Figure 4.2: Turbo coding BER performance using different numbers of iterations

The figure 4.3 illustrates the performance of turbo codes as a function of frame length. In many applications, particularly those involving real-time transmission, large frame lengths are impractical. For instance, frames of 256 bits are suitable for voice transmission, while frame lengths ranging from 1024 to 2048 bits are commonly used for video transmission. Systems employing larger frame lengths are better suited for data transfer and non-real-time applications. The simulation results indicate that turbo codes with a frame length of 65,536 bits achieve the best performance. Specifically, a frame length of 65,536 bits provides a code gain of 0.35 dB compared to turbo codes with a frame length of 2048 bits and 0.6 dB compared to codes with a frame length of 1024 bits, for a BER of 10^{-4}. As the frame length increases, the performance of turbo convolutional codes improves significantly.

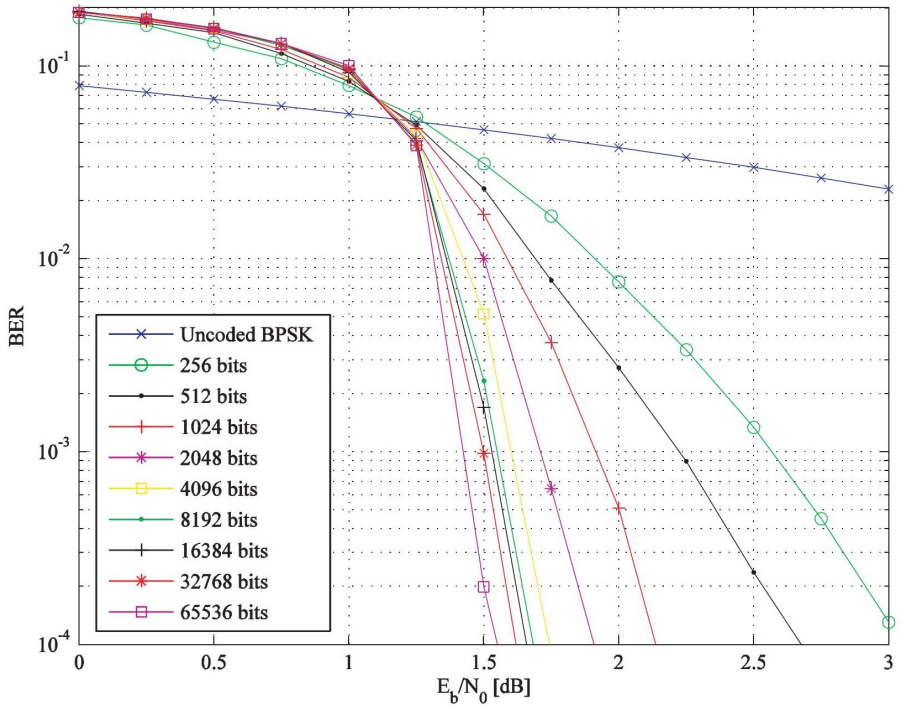


Figure 4.3: Effect of frame length on BER performance of turbo coding

Several different algorithms have been considered, but Turbo codes were deemed the most suitable for the application in question.

Table 4.1 shows that turbo codes offer an excellent solution for satellite communication systems due to their superior error-correction capabilities, low computational complexity, and ability to approach the theoretical limits of error correction. These properties make them particularly well-suited to the noise-rich, bandwidth-constrained environments of satellite communication and cryptographic applications, which rely on non-real-time interactions and involve larger frames, such as encryption keys.

The remarkable performance of turbo codes is primarily attributed to two fundamental principles: *iterative decoding* and *interleaving* between concatenated parallel codes. The effectiveness of turbo codes can be understood by examining the limitations of traditional error correction methods. In low SNR environments, error correction capabilities are constrained by the minimum distance properties of finite-length codes. Consequently, a single decoding pass may not be sufficient to correct all errors. A straightforward approach to enhancing performance would be to reapply the decoding algorithm multiple times. However, this method presents a significant drawback: it can introduce structured error patterns, such as burst errors, which are particularly prevalent in harsh transmission conditions like space

Category	BCH Code	Turbo Code	LDPC Code
Features	Fast decoding with limited error correction (up to 2 bits), 4/7 coding rate.	Parallel concatenation, 1/3 coding rate, 200 iterations with random interleaver.	1000 bits of information coded into 500 bits; uses belief propagation for decoding.
Performance	Fastest decoding speed among the three but limited correction capability.	Superior performance at low SNRs (below 8.1 dB) with high accuracy.	Outperforms Turbo Code at high SNRs, up to 8.1 dB.
Complexity	Low-complexity, suitable for low-error environments where speed is prioritized.	High complexity but suitable for environments with low SNR tolerance.	Less complex than Turbo, ideal for higher SNR applications.
Applications	Speed-critical, low-complexity applications with minimal error correction needs.	Low SNR applications needing high error correction performance.	High-efficiency, computationally optimized applications at higher SNRs.

Table 4.1: Performance Comparison of , Turbo, and LDPC Codes

and satellite communications. Rather than reducing errors, these patterns can propagate and degrade decoding performance, requiring an alternative strategy to ensure reliable data recovery.

To mitigate this issue, interleaving is introduced between successive decoding stages. By redistributing errors over a longer sequence, interleaving prevents burst errors from clustering, thereby enhancing the effectiveness of iterative decoding. This principle extends to the encoding stage as well, ensuring that the transmitted signal benefits from the same robustness against structured errors.

4.1 Turbo Encoding

The turbo code encoder is a fundamental component in modern error correction techniques, designed to achieve near-optimal performance in noisy communication

channels. Its structure consists of two identical recursive systematic convolutional (RSC) encoders arranged in parallel and separated by an interleaver. Each RSC encoder typically operates at a rate of $r = \frac{1}{2}$ and is referred to as a component encoder. To optimize efficiency, only one of the systematic outputs is retained, while the other is discarded, as it is merely a permuted version of the retained sequence.

In the figure below, a simplified diagram of a Turbo encoder structure is shown.

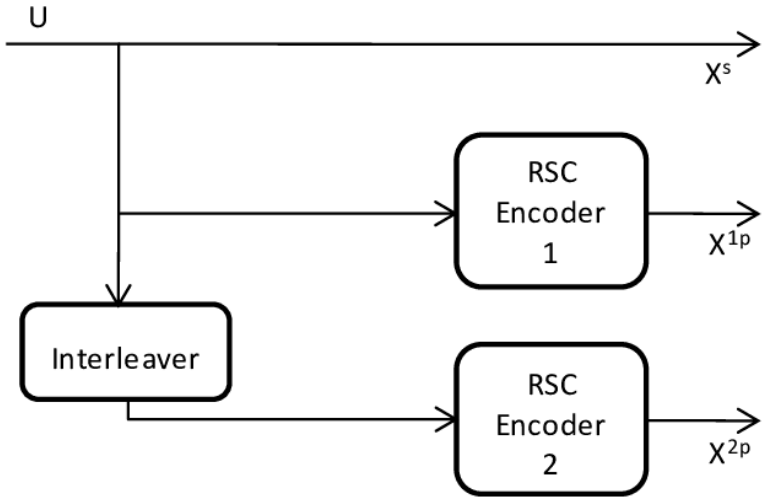


Figure 4.4: Fundamental Turbo Code encoder

A commonly used turbo encoder configuration operates at a rate of $r = \frac{1}{3}$, where the first RSC encoder produces a systematic output along with a recursive convolutional output, while the second RSC encoder suppresses its systematic sequence and provides only an additional recursive convolutional output. The inclusion of an interleaver enhances error correction performance by dispersing error patterns, thus improving the effectiveness of iterative decoding.

4.1.1 Recursive Systematic Convolutional Encoder

The recursive systematic convolutional (RSC) encoder is derived from a conventional convolutional encoder by introducing a feedback loop. This structural modification significantly enhances coding performance by increasing the randomness of the encoded sequences, thereby reducing error propagation and improving the overall robustness of the turbo code.

A conventional convolutional encoder is typically defined by a set of generator polynomials, which determine the transformation of input data into encoded sequences. The RSC encoder modifies this structure by feeding back one of its output

sequences into the input, effectively creating a recursive process. This feedback mechanism ensures that the generated sequences exhibit desirable properties such as long pseudo-random distributions, which are crucial for effective error correction. The use of an appropriate feedback polynomial further enhances performance by generating maximum-length sequences, maximizing the interleaving gain, and improving the distance properties of the code. In the figure below, a diagram of the RSC that highlights the recursive mechanism.

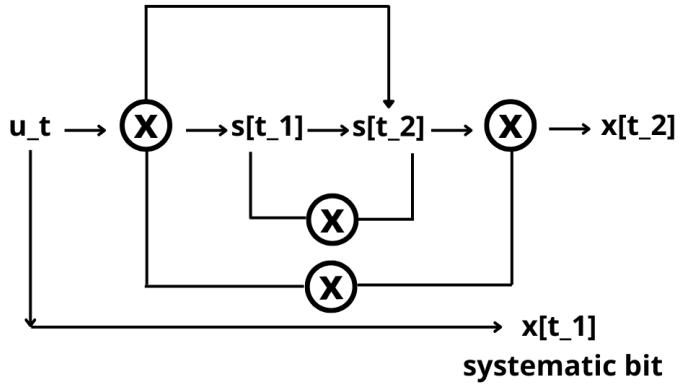


Figure 4.5: Recursive Systematic Convolutional encoder

As shown in the figure, the RSC encoder operates by maintaining a set of states and recursively combining these states with the current input. The encoder produces two types of outputs:

- **Systematic Output:** This output directly corresponds to the input bits, u_t, where t denotes the time index. It is typically transmitted as is in the output sequence: $xt_1(t)$.
- **Encoded Output:** The encoded output depends not only on the current input bit u_t but also on the previous states of the encoder. These outputs are denoted by xt_1 and xt_2, representing the encoded outputs that are generated by the combination of the input and the previous states:

$$xt_2(t) = g(st_1(t), st_2(t), u_t)$$

  where g is a function that combines the current input and the encoder states to produce the encoded output. The encoded output is typically used to create redundancy in the transmitted signal, providing error correction capabilities.

The encoder maintains two internal states, st_1 and st_2, which evolve over time based on the input sequence. The states and outputs at each time step t are updated recursively as follows. Let the input bit sequence be denoted by $u_0, u_1, u_2, \ldots, u_{N-1}$, where N is the total number of input bits. At each time step, the encoder generates two outputs: the systematic output and the encoded output. The internal states of the encoder, denoted by st_1 and st_2, evolve over time according to the following recurrence relation:

$$st_1(t) = f_1(st_1(t-1), st_2(t-1), u_t)$$

$$st_2(t) = f_2(st_1(t-1), st_2(t-1), u_t)$$

where f_1 and f_2 are functions that combine the previous states $st_1(t-1)$ and $st_2(t-1)$, along with the current input bit u_t, to determine the new state values at time step t. The states st_1 and st_2 serve as the memory of the encoder and are updated at each time step. To ensure the proper termination of the encoding process and finalize the encoded output, the encoder may continue processing with dummy input bits, often set to zero or a predefined value, after all valid input bits have been processed. This is known as the termination phase and ensures that all states are properly updated and the final encoded bits are produced. During the termination phase, the states evolve according to the same recurrence relations, and the final outputs are generated. The termination ensures that the encoder's state machine reaches a known final state, and all necessary encoded bits are produced. This memory allows the encoder to spread information across multiple output bits, increasing the error-correcting capability. The memory of the encoder ensures that it is not only reacting to the current input but also to the history of the transmitted data. This is important in the presence of noise or channel impairments, as errors tend to occur in bursts. By using the memory of previous states, the encoder can recover from errors that affect multiple bits in a burst.

4.1.2 Trellis Diagram

The *trellis diagram* is a fundamental representation used in convolutional coding to illustrate state transitions over time. Convolutional codes, widely employed in digital communication and error correction, encode data by generating output symbols based on the current input and a memory of previous inputs. The trellis diagram provides a structured visualization of the encoder's state evolution, where nodes represent states and branches correspond to valid transitions dictated by the convolutional generator polynomials. Formally, a convolutional encoder can be modeled as a finite-state machine, with its state determined by the contents of its shift registers. The trellis unfolds this state transition process over discrete time steps, creating a lattice-like structure. This representation is particularly useful in

decoding algorithms, most notably the Viterbi algorithm, which performs maximum likelihood sequence estimation by tracing the optimal path through the trellis. By assigning path metrics to transitions and employing dynamic programming techniques, the algorithm efficiently identifies the most probable transmitted sequence while minimizing the probability of error. Beyond its theoretical significance, the trellis diagram plays a critical role in practical implementations of convolutional codes, particularly in communication systems that require robust error correction, such as satellite communications and deep-space telemetry. Its structured nature allows for computationally efficient decoding, reducing the complexity compared to exhaustive search methods.

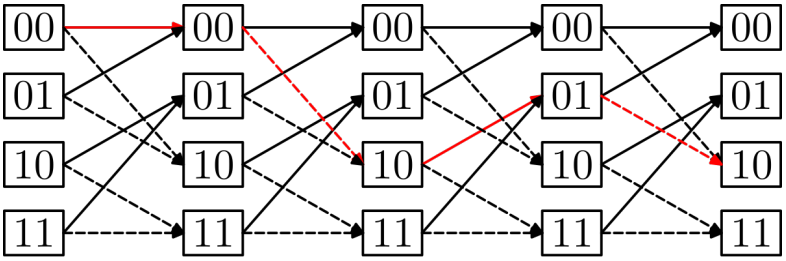


Figure 4.6: Trellis Diagram structure

In conclusion, the trellis diagram serves as both an analytical tool and a computational framework in convolutional coding theory. It enables effective error correction through structured state transitions and facilitates efficient decoding algorithms, making it an essential component in modern digital communication systems.

4.2 Interleaver Design

The interleaver plays a particularly crucial role in Turbo codes, as its design strongly influences system performance, particularly in the error floor region, where low-weight codewords contribute significantly to the bit-error rate (BER) and frame-error rate (FER). Consequently, an optimally designed interleaver is essential to mitigate performance limitations and suppress the error floor. A fundamental aspect of turbo codes is the presence of recursive constituent convolutional codes (CCs) combined with interleavers of specific lengths. While initial optimization efforts have primarily relied on randomly chosen interleavers, there is no systematic approach to interleaver design due to the inherent complexity of the problem. [22] presents a systematic approach to interleaver design, specifically tailored to given constituent convolutional codes (CCs), providing valuable insights into its effectiveness in the software implementation of the project. The design process is

influenced by several key parameters, including the characteristics of the constituent CCs, the operating signal-to-noise ratio (SNR), the interleaver length, and the decoding method. By optimizing interleaver construction with respect to these factors, the proposed methodology enhances the overall performance of turbo codes. This aspect is particularly crucial in space communications, where high burst error rates pose a significant challenge. The interleaver plays a fundamental role by acting on the input sequence, dispersing errors more uniformly, and thereby improving decoding efficiency. As a result of the widespread use of such algorithms, it has been demonstrated that a structured approach to interleaver design can yield substantial performance gains by enhancing the distance spectrum of the resulting code. For these reasons, the approach presented in [22] has been adopted in this thesis work.

4.2.1 Minimal-Delay Interleaving and Causality

The delay in the interleaving process is a critical parameter, especially in applications where latency must be minimized. The end-to-end interleaving-deinterleaving delay is defined as the time interval between the arrival of the first input symbol at the interleaver and the corresponding output symbol at the deinterleaver. A causal interleaver must ensure that no output symbol is produced before its corresponding input is received. The concept of cycles in permutations plays a key role in characterizing interleaver delays. Any permutation can be expressed as a product of disjoint cycles, with transpositions (cycles of length two) forming the fundamental building blocks.

4.2.2 Definition and Structure of the Finite-State Permuter (FSP)

The FSP is introduced as a model for implementing interleavers with minimal memory requirements. The FSP operates using a sliding window mechanism that transposes elements dynamically as the input sequence progresses. The delay associated with an interleaver is determined by the transposition with the largest span. Each permutation can be uniquely described by a transposition vector, which encodes the sequence of swaps necessary to achieve the desired output order. An example permutation is given as:

$$\pi = (4,1,3,2) \tag{4.1}$$

where the transposition vector is derived step by step. The delay of this permutation is determined by the largest transposition span. A modular implementation of the FSP is proposed, using a queue-based approach. The transpositions are applied sequentially, ensuring that elements are ejected in the correct order while

maintaining a minimal memory footprint. Given a transposition vector for a permutation π, the inverse permutation π^{-1} can be computed iteratively. The recursive inversion algorithm is formulated using a sequence of logical operations that reconstruct the original ordering.

4.2.3 Interleaver Construction

The design of interleavers is a fundamental aspect of turbo code optimization, as it directly influences the performance of parallel concatenated convolutional codes (PCCC). Specifically, the interleaver determines the mapping of input error sequences, which in turn affects the minimum distance of the code and the multiplicity of low-weight codewords, both of which are critical to the overall bit error rate (BER). Given the combinatorial complexity of interleaver selection, an exhaustive search across all possible permutations is computationally impractical. To address this challenge, the paper taken into consideration introduces an iterative interleaver growth algorithm that incrementally constructs optimized interleavers while maintaining polynomial computational complexity. To guide the interleaver design process, a cost function is introduced to evaluate the impact of different permutations on code performance. Consider a terminating error pattern of length l and Hamming weight w associated with a recursive systematic convolutional (RSC) constituent code. The interleaver of length N permutes these error patterns, thereby altering their distribution within the code structure. A simplified version of the cost function, denoted as $C(\pi, e)$, is defined as:

$$C(\pi, e) = \sum_{i=1}^{m} h(d_i, w_i), \tag{4.2}$$

where d_i represents the minimum Euclidean distance of the codeword after interleaving, and w_i is the corresponding input weight. A commonly employed formulation for $h(d, w)$ is:

$$h(d, w) = w \cdot e^{-d^2/(2\sigma^2)}, \tag{4.3}$$

where σ^2 denotes the noise variance. This function prioritizes configurations that maximize the minimum distance while suppressing low-weight codewords, as these contribute disproportionately to BER degradation. Rather than attempting to optimize an interleaver in a single step, the proposed approach constructs an interleaver progressively, expanding an initial configuration of size N to size $N + 1$ while ensuring that each modification minimizes the overall cost function. The algorithm proceeds as follows:

1. **Initialization:** Begin with a small interleaver, either derived from an exhaustive search for short sequences or selected from an established heuristic.

2. **Incremental Expansion:** At each step, a new element is introduced into the interleaver sequence. Multiple possible placements are evaluated, and the one that minimizes the cost function is selected.

3. **Termination:** The iterative process continues until the interleaver reaches the desired length N.

This method significantly reduces computational complexity compared to brute-force approaches while still achieving near-optimal results. The computational complexity of interleaver construction depends on the implementation of the cost function. A straightforward approach results in a complexity of $O(N^3)$, whereas an optimized version, leveraging probabilistic averaging over error patterns, achieves a more efficient $O(N^2)$. This reduction is crucial for practical applications, enabling the optimization of interleavers for large block lengths without excessive computational overhead. Experimental validation confirms that the interleavers designed using this approach yield significant performance gains, making them highly suitable for noise-limited communication environments.

4.3 Input Sequence Generator

Pseudorandom sequence generators based on polynomial generators play a critical role in fields such as cryptography, error-correction coding, and numerical simulations. This section provides an overview of the theory related to the cyclicity of the sequences, the logarithmic relationship to sequence length, and the criteria for generating pseudorandom strings.

Polynomial generators are used to construct Linear Feedback Shift Registers (LFSRs), which are one of the most widely used methods for generating pseudorandom sequences. An LFSR is a shift register consisting of n bits, where the input bit is calculated as a linear combination of previous bits, determined by a generator polynomial:

$$P(x) = x^n + c_{n-1}x^{n-1} + \cdots + c_1 x + 1$$

where the coefficients c_i belong to the finite field F_2 (i.e., they take values 0 or 1) and determine which bits contribute to the feedback. The sequence produced by an LFSR has a maximum length of $2^n - 1$ if and only if the generator polynomial is *irreducible* and *primitive* over F_2. The sequences generated by an LFSR are cyclic, meaning that after a certain number of steps, the register returns to a previously encountered state, and the sequence repeats. If the generator polynomial is primitive, the sequence will have the maximum possible period:

$$T = 2^n - 1$$

If the polynomial is not primitive, the period of the sequence will be less than $2^n - 1$ and could be a divisor of this value. The polynomials presents some common behaviours:

- Every sequence generated by an LFSR is deterministic and repetitive.

- If a non-zero initial seed is selected, the sequence will always have the same period and repeat the same values.

- The generator polynomials determine the behavior of the sequence and the distribution of its bits.

The use of the logarithm of the sequence length refers to calculating the minimum required register size to generate a sequence of a given length.

4.3.1 Assumption of Uncorrelated Input Bits

In the conducted project, it was initially assumed that the random bits generated were completely uncorrelated, allowing for the straightforward use of these bits as inputs for subsequent simulations and cryptographic operations. However, during the verification phase, graphical analyses and simulations were performed, comparing the public and private keys, as well as ciphertexts generated by the Kyber encryption scheme, with other random inputs derived from polynomial-based generators. The results revealed that the random bits were sufficiently independent, as correlations between the bits could not be found.

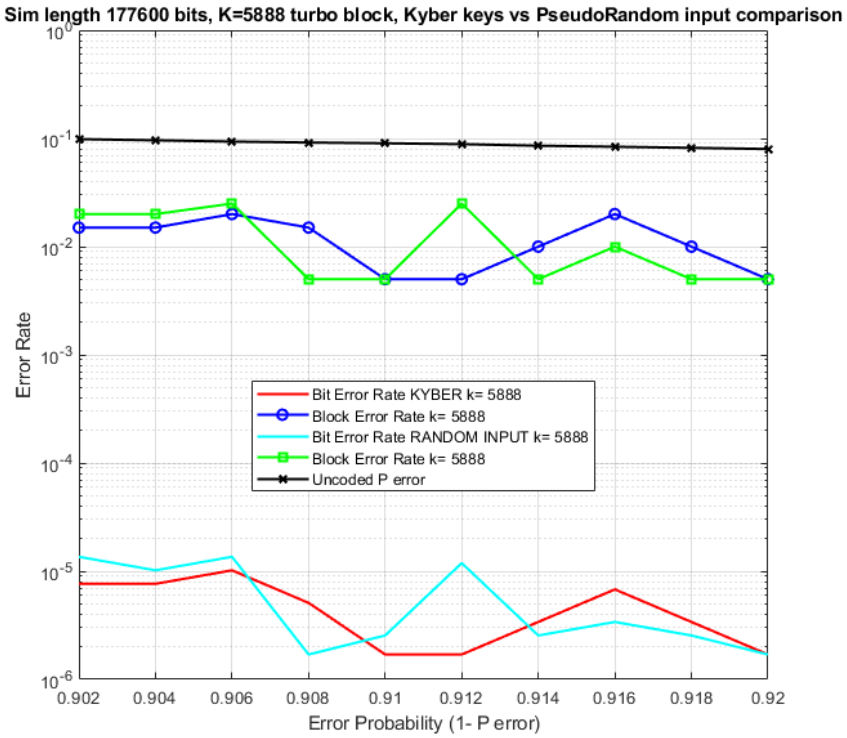


Figure 4.7: Simulation results with 2 different inputs: Kyber keys and Pseudo Random Input

As shown in Figure 4.7, the simulations run with different input sequences report the inability to reveal any correlation between bits. This finding is significant because the quality of random bit generation has a direct impact on the security and reliability of cryptographic systems. Specifically, when bits exhibit correlations or lack independence, the effectiveness of cryptographic algorithms, such as Kyber, may be compromised, as predictable patterns could be exploited by adversaries. Therefore, ensuring that generated bits are adequately uncorrelated is crucial for maintaining the robustness of cryptographic operations, statistical simulations, and security protocols, all of which rely on the assumption of high-quality randomness.

4.4 Turbo Decoding

A detailed overview of the turbo decoder structure is presented in [19], followed by the corresponding implementation in Rust programming language. The turbo decoder is an iterative decoding scheme that relies on two soft-input soft-output (SISO) decoders. Each decoder performs decoding in an iterative manner, improving

the estimation of the transmitted data sequence with each iteration. Initially, the first SISO decoder receives soft channel inputs and provides a soft output that estimates the original data sequence. In addition to the soft output, the decoder generates an extrinsic output, which is based on surrounding bits and the constraints imposed by the code, rather than directly on the channel input for that bit. This extrinsic information is then used as a priori information by the second SISO decoder, along with the channel inputs, to generate its own soft output and extrinsic information. This process is repeated in subsequent iterations, with each decoder receiving extrinsic information from the previous iteration and refining the decoded sequence. The turbo decoder iteratively improves the Bit Error Rate (BER) by leveraging both channel inputs and extrinsic information from previous decoding steps. After a fixed number of iterations, typically between 2 and 12, or when a stopping criterion is met, the decoding process terminates, and a final estimate of the transmitted sequence is provided. The iterative nature of the turbo decoder enables it to achieve near-Shannon-limit performance with relatively simple component codes, making it a powerful tool in error correction. The decoder's structure is clearly represented in the figure below.

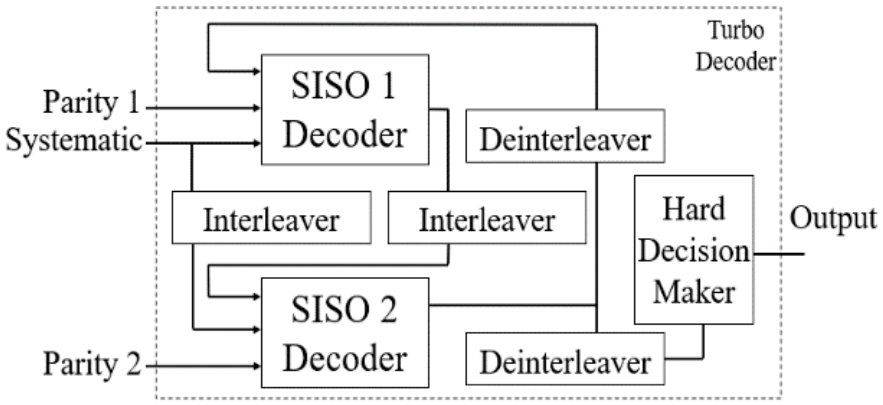


Figure 4.8: Turbo Decoder scheme

4.4.1 SISO Decoder

Soft-In Soft-Out (SISO) decoding is an essential technique in modern error-correcting systems, particularly in turbo and LDPC codes. Unlike conventional hard-decision decoders, which operate on binary values, SISO decoders utilize probabilistic (soft) information to enhance the reliability of decoded bits. This approach enables iterative refinement of bit estimates, significantly improving error correction performance.

The decoding process involves two fundamental operations:

- **Metric Combination:** Input likelihood values are aggregated to compute a reliability metric for each possible codeword configuration.
- **Marginalization:** The computed metrics are used to derive posterior probabilities for individual transmitted bits, considering all valid codeword configurations.

SISO decoders leverage iterative processing, where the extrinsic information from one iteration serves as *a priori* information for subsequent iterations. Depending on the decoding strategy, different computational approaches can be employed, such as the *min-sum* algorithm in the log-likelihood ratio (LLR) domain or the *sum-product* algorithm in the probability domain. These techniques balance complexity and performance, making SISO decoding highly effective in mitigating errors introduced by noisy communication channels.

The ability to process and refine soft information makes the SISO decoder a crucial component of iterative decoding architectures, offering superior performance over traditional hard-decision methods.

4.5 The BCJR Algorithm in Turbo Decoding

The BCJR algorithm, introduced in 1974 by Bahl, Cocke, Jelinek, and Raviv, is a decoding method based on *a posteriori* probabilities. Initially, its adoption in practical applications was limited due to its computational complexity. However, with the introduction of turbo codes in 1993 by Berrou, Glavieux, and Thitimajshima, a modified version of this algorithm became fundamental in iterative decoding techniques. The structure and the theory presented in [23] was crucial, in order to implement the turbo decoder in the Rust project.

A convolutional or block encoder can be represented using a trellis structure, which generates an output sequence $x = (x_1, x_2, \ldots, x_N)$ consisting of N symbols. Each transmitted symbol x_k is associated with an input bit u_k, which can take values ± 1 with an *a priori* probability $P(u_k)$. The reliability of these bits is expressed through the log-likelihood ratio (LLR):

$$L(u_k) = \ln\left(\frac{P(u_k = +1)}{P(u_k = -1)}\right). \tag{4.4}$$

For uniformly distributed input bits, this ratio is initially zero. When the sequence x is transmitted over an Additive White Gaussian Noise (AWGN) channel, the received sequence $y = (y_1, y_2, \ldots, y_N)$ is obtained. The BCJR algorithm estimates the original bit sequence by computing the *a posteriori* LLR:

$$L(u_k|y) = \ln\left(\frac{P(u_k = +1|y)}{P(u_k = -1|y)}\right). \tag{4.5}$$

The absolute value of $L(u_k|y)$ provides an indication of confidence in the estimated bit: the larger the magnitude, the higher the reliability of the decision.

4.5.1 Computation of Joint Probabilities

The BCJR algorithm operates by evaluating the joint probability of state transitions within the trellis. This probability is determined as follows:

$$P(s', s, y) = \alpha(s')\gamma(s', s)\beta(s), \tag{4.6}$$

where:

- $\alpha(s')$ represents the probability of reaching a given state s' based on past observations,
- $\gamma(s', s)$ denotes the probability of transitioning from state s' to s, given the received sequence,
- $\beta(s)$ indicates the probability of reaching a final state from s.

4.5.2 Recursive Computation of Alpha and Beta

The BCJR algorithm employs a recursive approach to efficiently compute these probabilities. The forward recursion (α) is given by:

$$\alpha_k(s) = \sum_{s'} \alpha_{k-1}(s')\gamma(s', s), \tag{4.7}$$

while the backward recursion (β) follows the expression:

$$\beta_k(s') = \sum_{s} \beta_{k+1}(s)\gamma(s', s). \tag{4.8}$$

These recursions allow the algorithm to propagate probability estimates both forward and backward along the trellis, refining the likelihood values used in the decoding process.

4.5.3 Computational Optimizations in MAP Decoding

Due to its inherent complexity, several approximations of the BCJR algorithm have been developed to facilitate practical implementations:

- **Log-MAP Algorithm**: This variation preserves the original MAP formulation but operates in the logarithmic domain, replacing multiplications with additions to enhance numerical stability.

- **Max-Log-MAP Algorithm**: A further simplification that approximates the logarithmic sum using a *maximum* operation, reducing computational complexity at the cost of minor performance degradation.
- **Soft-Output Viterbi Algorithm (SOVA)**: A modified version of the classical Viterbi algorithm that incorporates reliability information into the path metric without explicitly computing LLRs.

4.5.4 Application in Turbo Decoding

In turbo decoding, the BCJR algorithm serves as a core component. A systematic convolutional encoder generates multiple sets of parity bits, and decoding is performed iteratively using two MAP decoders. The *a posteriori* LLR is decomposed into different components:

$$L(u_k|y) = L_c y_k + L_a(u_k) + L_e(u_k). \tag{4.9}$$

The term $L_e(u_k)$, known as extrinsic information, is exchanged between the two decoders over multiple iterations, refining bit estimates progressively until a convergence criterion is met. This iterative information exchange significantly enhances decoding performance compared to non-iterative approaches.

4.6 Binary Symmetric Channel Model for Radiation Induced Errors

In conventional Soft-In Soft-Out (SISO) decoding, the channel is typically modeled as an Additive White Gaussian Noise (AWGN) channel, where Log-Likelihood Ratios (LLRs) are computed based on received symbols affected by Gaussian noise. However, in this implementation, the AWGN model has been replaced with a Binary Symmetric Channel (BSC) to simulate errors induced by radiation effects in memory systems.

In the BSC model, transmitted bits $x \in \{0,1\}$ undergo probabilistic flipping due to radiation-induced disturbances, leading to received bits $y \in \{0,1\}$. The transition probabilities are given by:

$$P(y = 0|x = 0) = 1 - p, \quad P(y = 0|x = 1) = p, \tag{4.10}$$

$$P(y = 1|x = 1) = 1 - p, \quad P(y = 1|x = 0) = p, \tag{4.11}$$

where p represents the probability of bit inversion due to radiation-induced errors. The corresponding LLRs are derived as:

$$LLR(y = 0) = \log\left(\frac{P(y = 0|x = 0)}{P(y = 0|x = 1)}\right) = \log\left(\frac{1 - p}{p}\right), \tag{4.12}$$

$$LLR(y=1) = \log\left(\frac{P(y=1|x=0)}{P(y=1|x=1)}\right) = \log\left(\frac{p}{1-p}\right) = -LLR(y=0). \quad (4.13)$$

Unlike the Gaussian noise model, where errors are continuous and follow a probabilistic distribution, the BSC model directly represents discrete bit flips, making it a more suitable approach for simulating the effects of radiation-induced memory corruption. The SISO decoding process remains unchanged in structure, utilizing the BCJR algorithm with forward-backward recursion, but now operates under this modified channel model to reflect the targeted error environment.

Chapter 5

Implementation of CRYSTALS-Kyber and Turbo Codes: A Modular Approach

5.1 Implementation Structure of CRYSTALS-Kyber

The project is structured to provide a modular and efficient implementation of the CRYSTALS-Kyber post-quantum key encapsulation mechanism (KEM) in Rust. The implementation is updated to the last NIST submission of Kyber, which is Round 3, from the official documentation. The core components are organized into multiple directories within the `src` folder, each responsible for different aspects of the cryptographic protocol.

The `functions` module contains essential operations necessary for encryption and key generation. This includes:

- `compress.rs`, which handles data compression techniques relevant to Kyber,
- `encode.rs`, responsible for encoding operations,
- `hash.rs`, which implements cryptographic hash functions,
- `ntt.rs`, managing Number Theoretic Transformations (NTT) to optimize polynomial arithmetic, and

- `utils.rs`, which includes auxiliary utility functions supporting various computations.

The `kem` module, defined by `mod.rs`, implements the key encapsulation mechanism, which is the core of CRYSTALS-Kyber. Similarly, the `pke` module contains functionalities for public-key encryption, ensuring secure key exchanges.

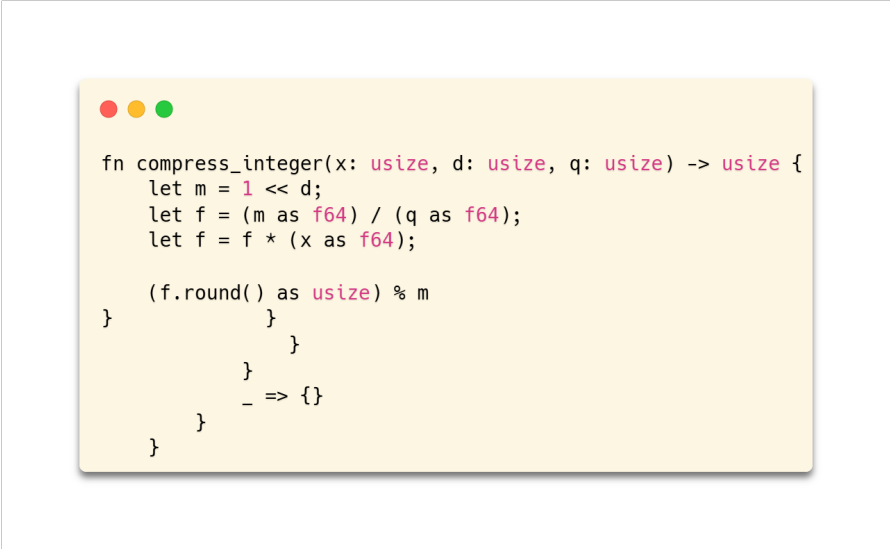
The `structures` directory encapsulates fundamental algebraic and data representations used in the cryptographic computations. Within this, the `algebraics` submodule defines:

- `mod.rs`, which provides module definitions,
- `bytearray.rs`, managing byte-level data structures for efficient data handling, and
- `primefield.rs`, which implements arithmetic operations over finite fields, crucial for lattice-based cryptography.

This organization ensures a clean separation of concerns, promoting reusability and efficiency while adhering to the principles of Rust's memory safety and performance optimization.

5.1.1 Compress and Encode functions

The provided function `compress_integer(x, d, q)` is a method for compressing an integer `x` into a smaller range, determined by the parameter `d`, while incorporating a scaling factor based on `q`. The function begins by calculating the value of `m`, which is defined as $m = 2^d$, representing the upper bound for the compressed result. This bound defines the range within which the compressed integer will fall. Next, the function computes a scaling factor `f` as the ratio $\frac{m}{q}$, which is used to proportionally adjust the input value `x`. The scaled value is then computed by multiplying `x` by `f`. To ensure the result remains within the desired range, the scaled value is rounded to the nearest integer, converted back to a `usize` type, and finally reduced modulo `m`. This ensures that the compressed value lies between 0 and $m - 1$. The approach effectively compresses `x` based on the specified scaling factor, providing a means of reducing its size while retaining proportionality to the parameters `d` and `q`.



```rust
fn compress_integer(x: usize, d: usize, q: usize) -> usize {
    let m = 1 << d;
    let f = (m as f64) / (q as f64);
    let f = f * (x as f64);

    (f.round() as usize) % m
}             }
                }
            }
            _ => {}
        }
    }
```

Figure 5.1: Kyber's Compress Function in Rust

The function `encode_poly`, implemented in Rust, is designed to encode a polynomial into a byte array representation, a key operation in the Kyber encryption scheme. It takes two parameters: a polynomial `p` of type `Poly3329<N>` and an integer `ell`, which specifies the number of bits per coefficient. The function iterates over the 256 coefficients of the polynomial `p`, converting each coefficient into its binary representation and encoding it into a byte array. Initially, an empty byte vector `b` is created, and a temporary variable `c` is set to zero, which will hold the bits of the current byte during the encoding process. For each coefficient `p[i]`, the function converts it to an integer using `to_int()`. Then, for each coefficient, it processes `ell` bits. Within each iteration, the bit value is checked, and if it is set (i.e., if the least significant bit is 1), the corresponding bit in the current byte `c` is updated. Once 8 bits have been processed (i.e., when `s == 0`), the byte `c` is appended to the vector `b`, and a new byte is initialized for the next group of bits. This process continues until all coefficients have been processed. Finally, the function appends the last byte to the vector and returns the byte array as `ByteArray::from_bytes(b.as_slice())`. This encoding method is crucial in the Kyber scheme as it allows efficient storage and transmission of polynomial coefficients in a compact binary format.

```rust
pub fn encode_poly<const N: usize>(p: Poly3329<N>, ell: usize) -> ByteArray {
    let mut b = vec![];
    let mut c: u8 = 0;

    for i in 0..256 {
        let mut v = p[i].to_int();
        for j in 0..ell {
            let s = (i * ell + j) % 8;
            if s == 0 && !(i == 0 && j == 0) {
                b.push(c);
                c = 0;
            }
            if (v & 1) == 1 {
                let a = 1 << s;
                c += a as u8;
            }
            v >>= 1;
        }
    }
    b.push(c);
    ByteArray::from_bytes(b.as_slice())
}
```

Figure 5.2: Kyber's Encoding Function in Rust

5.1.2 Number Theoretic Transform

The implementation of the Number Theoretic Transform (NTT) operates over the finite field $\mathbb{Z}_{3329}$ and efficiently enables polynomial multiplication in the NTT domain. The core components of the implementation include predefined constants, transformation functions, and multiplication routines. The array `ZETAS_256` stores the 256-th roots of unity required for the NTT, while the function `byte_rev(i)` serves as a placeholder for bit-reversal permutation, which is essential for ordering input coefficients. The primary data structures include `Poly3329<N>`, representing polynomials in $\mathbb{Z}_{3329}$, `PolyVec3329<N, D>`, for vectors of polynomials, and `PolyMatrix3329<N, X, Y>`, for polynomial matrices. The base case multiplication (BCM) routines perform direct multiplication in the NTT domain. The function `bcm()` executes polynomial multiplication by leveraging the precomputed roots of unity and a butterfly structure to update coefficients efficiently. The function `bcm_vec()` extends this operation to vectors by computing element-wise products and summing the results, while `bcm_matrix_vec()` applies the same principle to matrix-vector multiplications. The forward transformation is handled by `base_ntt()`, which maps a polynomial from coefficient space to the NTT domain. This function iteratively updates coefficients using modular arithmetic and the stored roots of unity. The function `ntt_vec()` extends this transformation to polynomial vectors by applying `base_ntt()` to each element. The inverse

transform is implemented through `rev_ntt()`, which restores polynomials to their original domain by applying inverse roots of unity and normalizing the coefficients by $(d/2) + 1$. The function `rev_ntt_vec()` extends this operation to vectors by applying `rev_ntt()` to each component. Polynomial multiplication in the NTT domain is performed using `ntt_product()`, which computes a pointwise product of two polynomials in the transformed domain, followed by an inverse NTT to recover the final result. The functions `ntt_product_vec()` and `ntt_product_matvec()` generalize this operation to vectors and matrix-vector products, respectively. To ensure correctness, the implementation includes two validation tests. The function `rev_then_ntt()` verifies that applying the inverse NTT followed by the forward transform correctly restores the original polynomial. Applying `ntt_then_rev()` guarantees that performing the NTT followed by its inverse reproduces the original input polynomial, thereby verifying that the transformation is both consistent and reversible.

5.1.3 Key Encapsulation Mechanism (KEM)

The structure `KEM<N, K>` encapsulates the underlying Public Key Encryption (PKE) scheme and defines essential parameters such as key sizes, ciphertext size, and security parameters.

The `keygen()` function follows Algorithm 7 of the Kyber specification, generating a public-secret key pair. It first derives a random seed `z` and invokes the PKE's `keygen()` to generate `sk_prime` and `pk`. Hash values `h1` and `h2` of the public key are computed using the hash function `h()`, and the final secret key is constructed by concatenating `sk_prime`, `pk`, `h1`, `h2`, and `z`.

The `encaps()` function (Algorithm 8) generates a ciphertext and shared key based on the recipient's public key. A random message `m` is hashed, and the function `g()` is applied to derive a key component `k_bar` and randomness `r`. The ciphertext `c` is generated using the PKE encryption function. A key derivation function (`kdf()`) then derives the final shared key from `k_bar` and a hash of the ciphertext.

The `decaps()` function (Algorithm 9) retrieves the shared key from the ciphertext and secret key. The secret key is split into components: `sk_prime`, `pk`, a hash of `pk`, and `z`. The ciphertext is decrypted to obtain `m`, and the same derivation process as in `encaps()` is performed to recompute `k_bar` and a new ciphertext `c_prime`. If `c_prime` matches the received ciphertext, the correct shared key is derived; otherwise, a fallback mechanism using `z` ensures resistance against chosen-ciphertext attacks.

The implementation includes test cases verifying the correctness of key generation, encapsulation, and decapsulation for different security levels (Kyber512 and Kyber768). The tests ensure that after encapsulation and decapsulation, the shared keys remain consistent, confirming the correctness of the implementation. The

Rust implementation effectively follows the Kyber specification while leveraging modularization and type safety for efficiency and security.

5.1.4 Public Key Encryption (PKE)

The structure `PKE<N, K>` encapsulates the cryptographic parameters such as modulus `q`, noise distribution parameter `eta`, and compression parameters `du` and `dv`, which vary across different security levels.

The `keygen()` function (Algorithm 4, p. 9) generates a public-secret key pair. It first derives a 32-byte random seed `d`, from which two auxiliary values, `rho` and `sigma`, are computed via the function `g()`. The public matrix `A` is deterministically generated from `rho` using an extendable output function (`xof()`), ensuring security against key reuse attacks. The secret and error polynomials, `s` and `e`, are sampled from a centered binomial distribution using the function `cbd()`, ensuring small norm constraints. After applying the Number Theoretic Transform (NTT) to obtain `s_hat` and `e_hat`, the public key is computed as $\hat{t} = A \cdot \hat{s} + \hat{e}$. The secret key is stored as `s_hat`, while the public key consists of `t_hat` and `rho`.

The `encrypt()` function (Algorithm 5, p. 10) generates a ciphertext from a given public key, message, and randomness. The public key is split into `t` and `rho`, and the matrix `A_t` is regenerated using `xof()`. The randomness `r` is used to sample ephemeral polynomials, which are transformed via NTT to obtain `r_hat`. The ciphertext components are computed as $u = A^T \cdot \hat{r} + e_1$ and $v = t^T \cdot \hat{r} + e_2 + \text{Decompress}(m)$, where `e1` and `e2` are noise terms ensuring security. The ciphertext is finally compressed using `compress_polyvec()` and `compress_poly()`.

The `decrypt()` function (Algorithm 6, p. 10) recovers the plaintext from a given ciphertext and secret key. The ciphertext components are decompressed and the secret key is retrieved. Using NTT, the intermediate polynomial $x = s^T \cdot \hat{u}$ is computed, and the message is reconstructed as $p = v - x$. The plaintext is recovered via decompression and output in its encoded form.

The implementation includes test cases validating key generation, encryption, and decryption for Kyber512 and Kyber768 security levels. The encryption-decryption consistency test ensures that decrypting an encrypted message yields the original plaintext, demonstrating correctness and compliance with the Kyber standard.

5.1.5 Integration Testing

Some standard integration tests have been written, in order to verify the correctness of encryption and decryption functions in the Kyber implementation by simulating

a real-world use case. The tests encompass both the Key Encapsulation Mechanism and Public Key Encryption functionalities.

For KEM, the test follows the standard key exchange protocol: first, Alice generates a key pair (`sk`, `pk`) using `keygen()`, keeping `sk` secret while publishing `pk`. Bob uses `encaps()` with Alice's public key to derive a shared secret `k` and an encapsulated ciphertext `c`, which he sends to Alice. Alice then applies `decaps()` with her secret key to recover `k`, ensuring correctness via `assert_eq!(k, k_recovered)`.

For PKE: first, Bob generates a random message `m` and randomness `r`. Alice produces a key pair via `keygen()` and shares `pk`. Bob encrypts `m` with `encrypt()` using `pk` and `r`, storing the ciphertext. The ciphertext is then appended to a file, simulating real-world storage or transmission. Alice decrypts it with `decrypt()` using `sk`, verifying correctness with `assert_eq!(m, dec)`. This process ensures that decryption consistently retrieves the original message across multiple runs, validating both functional correctness and security compliance with the Kyber standard.

5.2 Implementation Structure of Turbo Codes

The `turbof` directory in the project contains the core implementation of Turbo Codes in Rust, structured to ensure modularity and maintainability. The module is organized into multiple files, each handling a specific aspect of the encoding and decoding processes. The primary components include `bsc_channel.rs`, responsible for simulating a binary symmetric channel, and `rsc_encoder.rs`, which implements a recursive systematic convolutional (RSC) encoder. The decoding functionality is split across `siso_decoder.rs` and `turbo_decoder.rs`, where the former manages soft-input soft-output decoding, while the latter coordinates the iterative decoding process. Additionally, `turbo_utils.rs` provides functions that handle alpha and beta recursion, random input sequence and other operations on ByteArrays. `turbo_simulation.rs` has a crucial structure, since it handles the error counts on bits and blocks, the simulation length in turbo-blocks and all the parameters applied to the algorithm, in order to analyze performance. In `tests/integration_test.rs` some integration tests are provided, in order to control the flow of the whole encryption-decryption operation employed with the error correction procedure. This modular design allows for efficient experimentation with different encoding and decoding strategies while maintaining a clean separation of concerns within the implementation.

5.2.1 Generating Input Sequence

The function `generate_pn_sequence` implements a pseudo-random number (PN) sequence generator based on a Linear Feedback Shift Register (LFSR). This function takes as input two parameters: `ndeg`, representing the degree of the polynomial, and `ngen`, which selects the specific set of polynomial coefficients to be used. The implementation is structured as follows.

Initially, three predefined sets of polynomial coefficients (`pol1`, `pol2`, and `pol3`) are defined in octal form. The selection of the polynomial set is determined by the `ngen` parameter. The `dig` vector provides a mapping between the polynomial degree and the number of active coefficients used in the sequence generation.

The function then constructs the characteristic polynomial `polcn` through an iterative process, where each coefficient is expanded according to predefined rules that map its value to specific bit patterns. This polynomial is used to determine feedback connections in the LFSR.

The sequence generation process starts by initializing the LFSR with all ones. For each iteration, a new bit is computed as a function of previous bits, following the feedback structure dictated by `polcn`. The final sequence, extracted from the LFSR evolution, contains $2^{ndeg} - 1$ elements, ensuring maximal-length sequences given appropriate polynomial choices.

This implementation leverages recursion and bitwise operations to efficiently compute sequences, which are widely used in cryptographic protocols and error correction coding.

5.2.2 Interleaver Functions

Several functions are defined in the project, for interleaving and deinterleaving sequences of integers and floating-point numbers. The primary functions, `mapint` and `mapint_f64`, perform interleaving operations on integer and floating-point vectors, respectively, while `mapdint` and `mapdint_f64` execute the corresponding deinterleaving processes.

The interleaving process is structured around a permutation vector that determines the reordering of elements within a data block. The algorithm partitions the input sequence into smaller blocks of size determined by the permutation vector, iteratively distributing elements based on predefined patterns. Similarly, the deinterleaving functions reverse this process, reconstructing the original ordering of elements from the interleaved structure.

The implementation leverages vector indexing and slicing to manipulate subarrays dynamically. Placeholder elements are temporarily inserted to handle indexing consistency, and conditional logic ensures correct assignments based on specific permutation rules. The algorithm also incorporates range checks to prevent out-of-bounds access, ensuring robustness in data handling.

These functions are crucial in communication systems and data processing, where interleaving is employed to mitigate burst errors and enhance error correction performance. The structured yet flexible design of the Rust implementation enables efficient and scalable interleaving and deinterleaving operations.

5.2.3 Turbo Encoder

The `TurboEncoder` is defined as a struct that encapsulates the core components necessary for turbo encoding, a powerful error-correction technique widely used in digital communication systems. The structure consists of an input sequence, a permutation vector, and two Recursive Systematic Convolutional (RSC) encoders. The constructor initializes the encoder by instantiating two RSC encoders, ensuring that they match the length of the input sequence. The `encode` function performs the encoding process in three main steps: first, the input sequence is encoded by the first RSC encoder, producing both systematic and parity bits. Then, the interleaved version of the input sequence is generated using a mapping function, which applies the permutation vector. Finally, the permuted sequence is encoded by the second RSC encoder, yielding another set of parity bits. The function ultimately returns the original systematic bits, the interleaved sequence, and the parity bits from both encoders, forming the complete turbo-coded output.

RSC Encoders

The `RSCEncoder` struct represents a convolutional encoder with memory, maintaining an internal state vector that evolves based on the input sequence. The encoding functions, `encode1` and `encode2`, follow a structured approach: they iteratively update the encoder's internal state and compute parity bits while ensuring systematic bit extension and trellis termination. The first encoder operates directly on the input, while the second processes a permuted version of the sequence to introduce interleaving.

5.2.4 Turbo Decoder

The Turbo Decoder is implemented as a Rust struct, encapsulating the necessary components for iterative decoding. It includes two instances of the SISO decoder and vectors to store the received signals, extrinsic information, and interleaved data. The decoder is initialized with the received sequences, coding parameters, and an interleaver permutation. Upon initialization, the received sequence undergoes a simulated transmission through a **Binary Symmetric Channel (BSC)**, where errors are introduced probabilistically. The decoder computes *a priori* log-likelihood ratios (LLRs) for systematic and parity bits, serving as inputs for the iterative

process. The decoding algorithm follows a sequence of iterative exchanges between two SISO decoders:

1. The first SISO decoder processes the systematic and parity inputs, generating an **a posteriori probability (APP)** estimate.
2. The extrinsic information from the first decoder is **interleaved** and provided as an input to the second SISO decoder.
3. The second SISO decoder processes the interleaved information, producing refined estimates.
4. The updated extrinsic information is **deinterleaved** and fed back into the first decoder.
5. This iterative process continues for a predefined number of iterations, refining the estimated transmitted sequence.

The decoder utilizes an **interleaver** to permute extrinsic information between iterations. The permutation function is applied after the first decoding step and reversed before feedback into the first decoder. This permutation enhances the decoder's ability to correct burst errors by redistributing them across different iterations. At each iteration, the decoder evaluates the **bit error rate (BER)** by comparing the estimated sequence with the transmitted sequence. The process stops when a predefined number of iterations is reached or when the BER stabilizes, indicating convergence. This Rust-based Turbo Decoder implementation efficiently applies **iterative decoding** using SISO decoders and interleaving mechanisms to enhance error correction performance. By exchanging soft information between decoders, the system improves the reliability of received messages, making it suitable for applications in modern digital communication systems.

SISO Decoder

The `SISODecoder` is implemented as a Rust struct, encapsulating the logic for decoding based on log-likelihood ratio (LLR) computations. The core of the decoding process consists of three main steps:

1. **Forward recursion** (Alpha computation)
2. **Backward recursion** (Beta computation)
3. **A posteriori probability (APP) computation** for bit estimation

The **Alpha recursion** computes the forward probability for each state transition in the trellis.

- Initialization is performed by setting initial state probabilities.
- At each time step, state probabilities are updated based on previous states and transition metrics.
- Normalization ensures numerical stability.
- The recursion extends through the entire received sequence and incorporates **trellis termination conditions**.

The **Beta recursion** follows a similar approach but operates in reverse:

- Initialization starts from the end of the sequence, assuming the final state probabilities.
- The backward pass iterates to determine the probabilities of each state at previous time steps.
- To improve efficiency, the recursion is processed in **blocks**, ensuring optimized memory handling.

Once the alpha and beta values are computed, the decoder determines the most likely transmitted bits:

- Log-likelihood ratios (LLRs) are computed for each bit using the **extrinsic information** and trellis states.
- The **maximum likelihood** decision rule is applied to estimate the transmitted bit sequence.
- A **thresholding mechanism** is used to finalize the decision for each bit.

To evaluate decoding performance, the decoder compares the estimated bit sequence with the original transmitted sequence:

- A **bit error count** is computed by comparing the estimated bits with the reference.
- The resulting metric allows for assessing decoder performance in iterative decoding scenarios.

The Rust-based **SISO Decoder** effectively applies iterative probability-based decoding to refine received signals. The combination of **forward recursion, backward recursion, and APP computation** enables the decoder to provide highly reliable bit estimates, making it a key component in Turbo Decoding architectures.

5.3 User Guide

This section is occupied by the readMe file written on the Github repository, in order to guide the user.

5.3.1 Read Me

This project implements **CRYSTALS-Kyber**, a *post-quantum cryptographic algorithm*, along with **Turbo Codes**, an advanced *error correction technique* used in communication systems.

- **Kyber** is a *lattice-based key encapsulation mechanism (KEM)* that provides secure encryption resistant to quantum attacks.
- **Turbo Codes** use *Recursive Systematic Convolutional (RSC) encoders* and iterative decoding to achieve near-optimal error correction performance in noisy channels.

The goal of this project is to implement a simulation of a **Kyber post-quantum encryption scheme**, to be used in **satellite communication in space**. Rust is the new programming language frontier and is now considered the future of space missions. The use of Turbo Codes allows for detecting and correcting errors caused by **radiation in space**, providing high performance.

5.3.2 Project Structure

The repository is structured into two main components:

Kyber Implementation (`src/kcimpl/`)

This folder contains the **Kyber cryptographic implementation**.

`kem/`

- `mod.rs`: Main module for the **Kyber Key Encapsulation Mechanism (KEM)**.

`functions/`

- `mod.rs`: Module handling.
- `compress.rs`: Implements Kyber's compression function, as defined in the official documentation.

- `encode.rs`: Implements Kyber's encoding function.
- `hash.rs`: Employes hash functions from the **SHA-3 standard**.
- `ntt.rs`: Implements the **Number Theoretic Transform (NTT)**.
- `utils.rs`: Implements helper functions such as hash and PRF.

`pke/`

- `mod.rs`: Implements **Public Key Encryption (PKE)** using Kyber.

`structures/`

- `algebraics/`: Contains mathematical structures used in Kyber.
  - `bytearray.rs`: Handles byte-level operations.
  - `primefield.rs`: Implements operations over finite fields, crucial for cryptographic computations.

5.3.3 Turbo Codes Implementation (`src/turbof/`)

This folder contains the **Turbo Codes implementation** for error correction.

- `bsc_channel.rs`: simulates a **Binary Symmetric Channel (BSC)**, introducing controlled errors for testing.
- `mapints.rs`: implements **interleaving functions** to permute input bits and improve error correction.
- `rsc_encoder.rs`: implements the **Recursive Systematic Convolutional (RSC) Encoder**, the core building block of Turbo Codes.
- `siso_decoder.rs`: implements the **Soft-Input Soft-Output (SISO) Decoder** for iterative decoding.
- `turbo_encoder.rs`: implements the **Turbo Encoder**, which consists of two RSC encoders and an interleaver.
- `turbo_decoder.rs`: implements the **Turbo Decoder**, performing iterative decoding for error correction.
- `turbo_simulation.rs`: runs simulations to evaluate Turbo Codes' performance under different channel conditions.
- `utils.rs`: provides helper functions for Turbo Codes processing.

5.3.4 Other Files

Test Files (`tests/`)

- `integration_test.rs`: Contains integration tests for Turbo Codes and Kyber implementations.

Data Files (`.txt`)

- `input.txt`: Example input data.
- `interleaver.txt`: Defines **interleaving patterns** for Turbo Codes.
- `kyber_keys.txt`: Stores cryptographic keys used in the Kyber algorithm.
- `output.txt`: Stores probabilities after correction, number of errors, bit and block error rate, and the real uncoded probability.

Kyber Test (`.rs`)

- `lib.rs`: Stores tests for Kyber KEM and PKE schemes. It can be used to simulate communication between Alice and Bob.

5.3.5 How to Use

In `lib.rs`, the user can configure the **CRYSTALS-Kyber simulation**, choosing the complexity of the algorithm and running tests for KEM and PKE.

In `integration_tests`, various tests exist to validate the Turbo Code algorithm. The function `test_turbo_simulation_dyn()` is the main test, running multiple simulations to gather statistical data.

Modifiable parameters:

- `simulation_length`
- `block_size`
- `error_probability`

The interleaver is set to the same length as the turbo-block. The interleaver is read from the file `interleaver.txt` (64,000 bits length). The results of simulations are stored in the `output.txt` file.

By assuming that the randomly generated input consists of uncorrelated bits, there is no difference between using a random sequence and using Kyber's keys (public key, secret key, or ciphertext). However, in `turbo_simulation.rs`, the code for using Kyber keys as input is commented out and can be activated at lines 81-82 by using an `input.txt` file.

Chapter 6

Results

6.1 Introduction

In this chapter, a comprehensive analysis of the Rust implementation is presented, focusing on two key aspects: the performance evaluation of the *Crystals-Kyber* algorithm and the simulation outcomes of *Turbo Codes*. The first section examines various performance metrics, including computational efficiency, key generation time, encryption/decryption latency, and security robustness, derived from the experimental data. The results are presented through graphical representations that illustrate the code's behavior over time after a sufficient number of executions. In the second section, the simulated behavior of Turbo Codes under different signal-to-noise ratio (SNR) conditions and error probabilities is analyzed, evaluating their error correction capabilities and decoding latency. The graphical analysis enables a comparative study of the efficiency of Turbo Codes in high-noise environments and their potential integration with post-quantum encryption schemes. The extensive literature on Turbo Codes allows for a comparison with existing studies, validating the current results and ensuring their consistency with expected performance. Ultimately, this project aims to develop a functional and efficient Rust implementation, providing a solid foundation for further optimization and real-world deployment.

6.2 CRYSTALS-Kyber Performance Analysis

The performance evaluation of the Rust implementation of the *Crystals-Kyber* algorithm, as illustrated in the provided figures, highlights key aspects of computational efficiency. The first graph, depicting CPU usage normalized per iteration, shows an initial peak followed by a rapid stabilization around 20%. This trend suggests that the system experiences an initial overhead, likely due to resource allocation and other optimizations inherent in Rust's execution model. As execution

progresses, the CPU utilization remains relatively stable with minor fluctuations, indicating a consistent and predictable computational demand. The normalized view is fundamental for analyzing CPU usage independently of the underlying hardware.

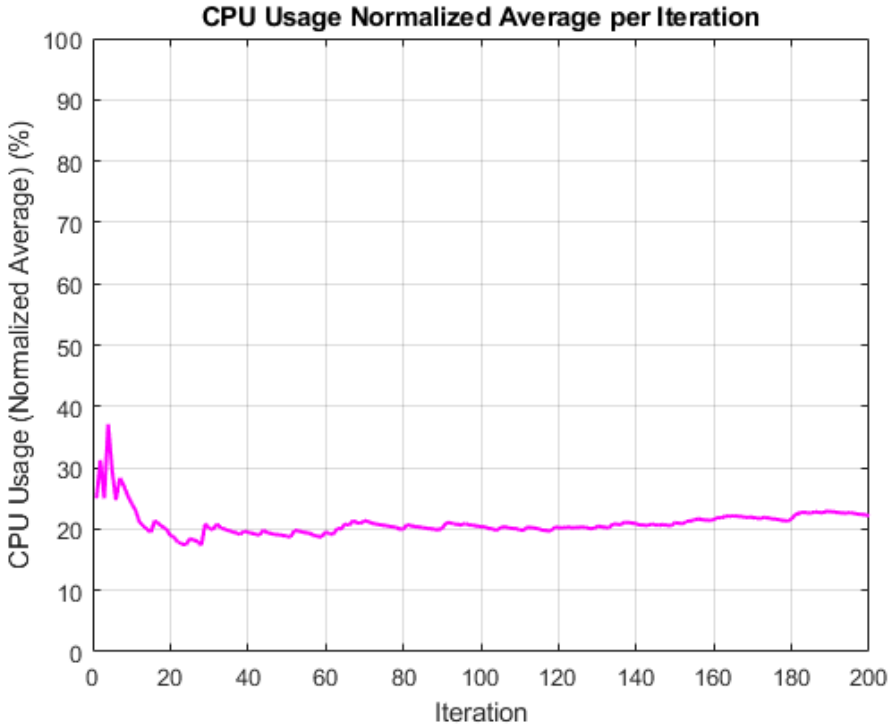


Figure 6.1: CPU Usage Normalized Average per Iteration

Figure 6.2 shows the raw CPU utilization based on the actual cores employed by the machine.

Figure 6.3, illustrating execution time per iteration, exhibits an average runtime oscillating between 19 and 22 milliseconds, with moderate variability. The decreasing trend in execution time fluctuations over iterations suggests an optimization effect, potentially attributed to caching mechanisms and branch prediction improvements. Overall, these results confirm that the Rust-based implementation achieves a balance between computational efficiency and execution stability, demonstrating its suitability for post-quantum cryptographic applications. Further optimizations could focus on reducing execution time variability to enhance performance consistency, particularly in latency-sensitive scenarios.

The memory usage per iteration is depicted in figure 6.4. The graph shows a significant peak in memory consumption at the initial iteration, reaching approximately 1.2×10^6 KB. This behavior suggests a high allocation cost at the

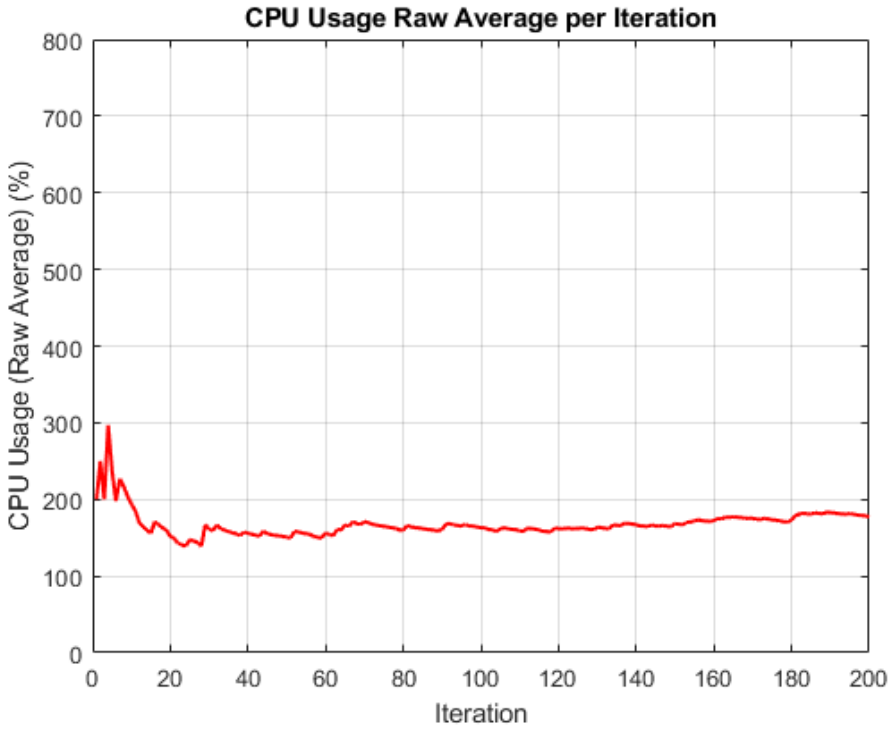


Figure 6.2: Raw CPU Utilization per Iteration

beginning of the execution, possibly due to the initialization of large data structures such as polynomials, key pairs, or precomputed tables. After this initial peak, the memory usage stabilizes at a much lower level, with occasional small spikes. These fluctuations could be attributed to the dynamic allocation and deallocation of temporary buffers or the execution of cryptographic transformations. The overall trend indicates that, once the fundamental structures are set up, the implementation maintains a relatively low memory footprint, which is crucial for resource-constrained environments such as embedded and space applications.

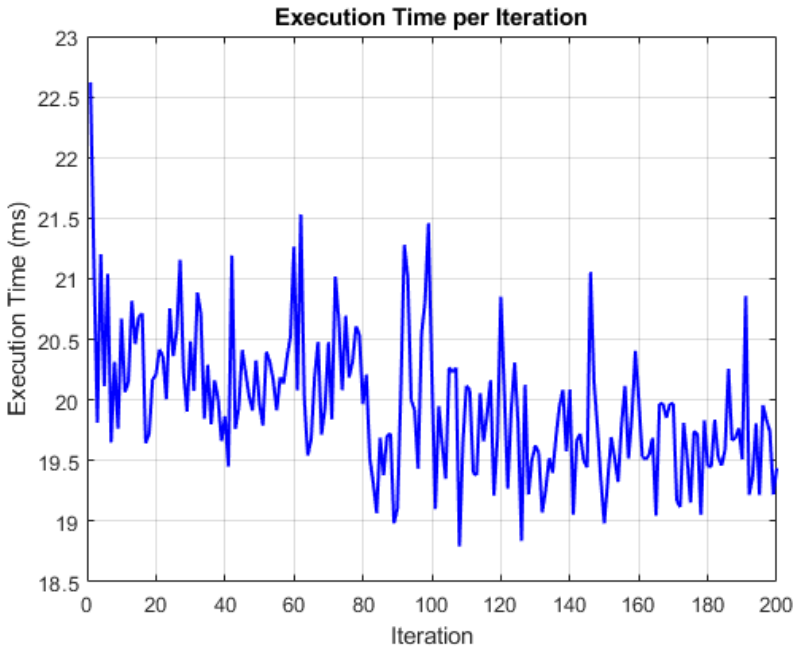


Figure 6.3: Execution Time per Iteration

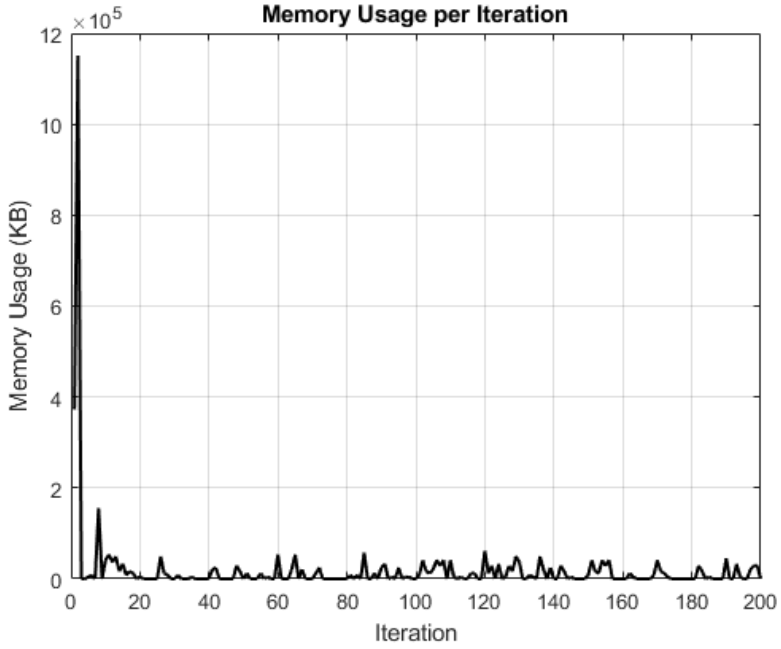


Figure 6.4: Memory Usage per Iteration

6.3 Turbo Code Performance Analysis

This section presents the performance analysis of Turbo Codes implemented in Rust, focusing on the impact of varying interleaver lengths (K) on bit error rate (BER) as a function of error probability ($1 - P$). The results are visualized in the following graphs. The findings highlight the coding gain achieved with increasing block lengths and the substantial improvement over uncoded transmission.

The key parameters are:

- **Total simulated block length:** 1,000,000 bits
- **Interleaver lengths (K):** $\{6400, 10000, 16000, 32000, 64000\}$
- **Channel:** Binary symmetric channel (BSC) with varying error probability $(1 - P)$
- **Decoding:** Log-MAP iterative decoding
- **Performance metric:** Bit Error Rate (BER)

Figure 6.5 illustrates the BER performance of Turbo Codes as a function of error probability $(1 - P)$ for different values of K. The key observations are:

- As K increases, the BER performance improves significantly, particularly in the high reliability region ($1 - P > 0.85$). This is expected, as larger interleaver sizes provide better randomness, reducing error propagation in iterative decoding.
- The uncoded bit error rate remains consistently high across all error probabilities, demonstrating the necessity of coding.
- For small values of K, the Turbo Code provides limited coding gain. However, as K increases to 64000, the BER drops to approximately 10^{-4}, highlighting the advantages of long interleavers.
- The waterfall region, where the BER rapidly declines, occurs at approximately $1 - P \approx 0.86$ for large K, p=1.4, aligning with typical Turbo Code behavior. This value of p is significantly higher than the error rate induced by space radiations, showing that the proposed scheme is able to protect data stored in a satellite memory from errors due to space radiations.
- As expected, there is a small region around $1 - P \approx 0.81$ where the Turbo Code is no longer able to correct errors efficiently. This occurs because the decoding algorithm relies on previously received information, which becomes less reliable in this regime.

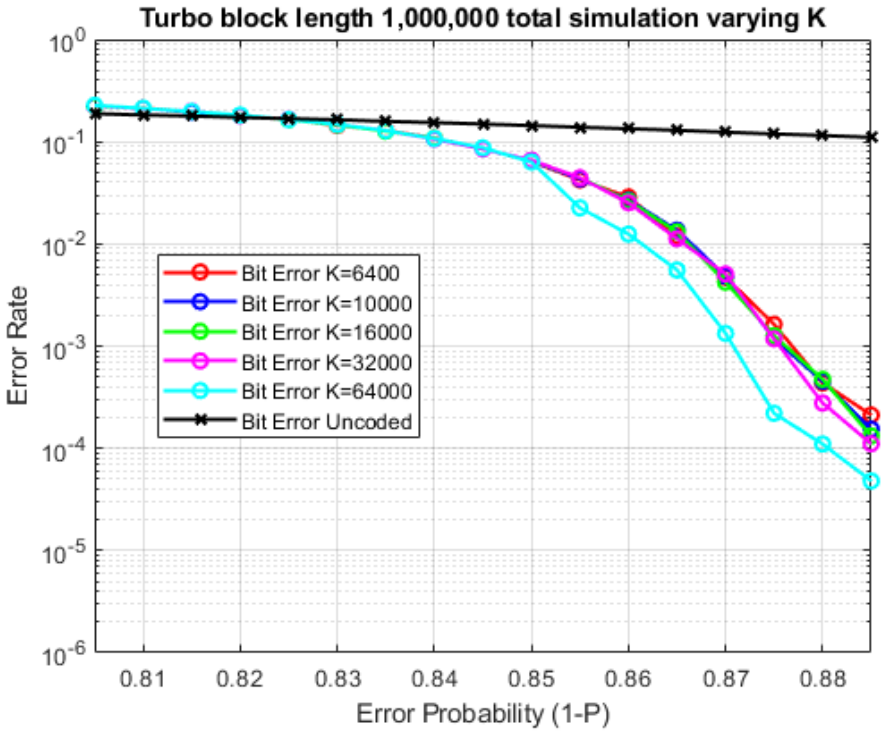


Figure 6.5: Simulation with same input sequence varying K

The figure 6.6 illustrates the BER and BLER for an interleaver length of $K = 64000$ as a function of the error probability P. As expected, the BER (red curve) decreases significantly for $P > 0.86$, indicating the strong error-correcting capability of Turbo Codes in this region. The BLER (blue curve) follows a similar trend but remains consistently higher, as an entire block must be error-free to avoid a block error. The uncoded error probability (black curve) remains nearly constant across all values of P, underscoring the necessity of channel coding.

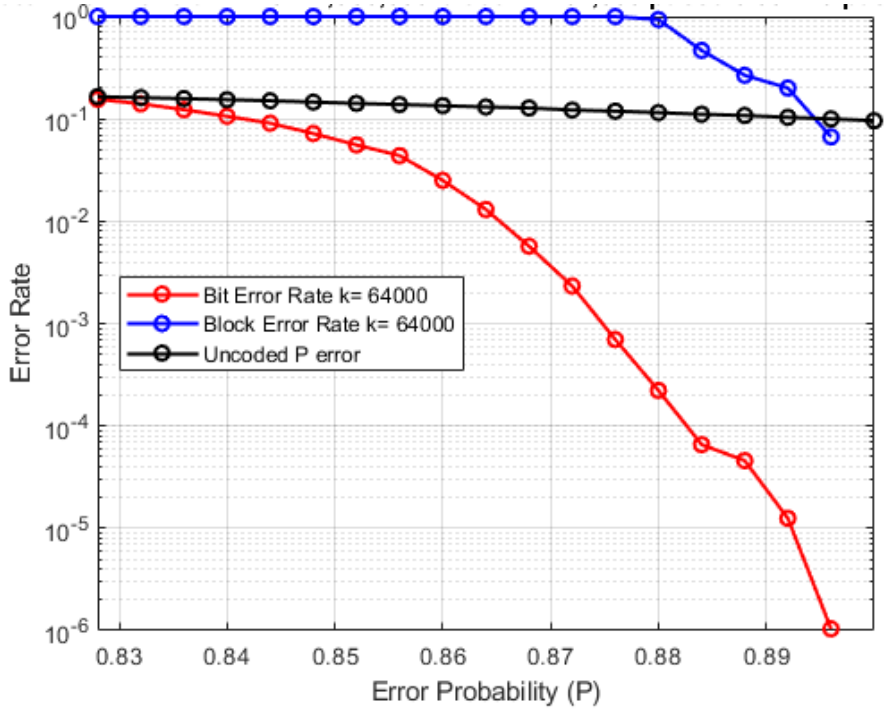


Figure 6.6: Bit Error Rate and Block Error Rate for Turbo Codes with interleaver length $K = 64000$

The figure 6.7 examines BLER for varying interleaver lengths K. As K increases, the performance improves, with a more pronounced waterfall region occurring at higher values of $1 - P$. This behavior confirms that larger interleavers enhance randomness in the encoded sequence, facilitating better iterative decoding performance. For small values of K, BLER remains high even in the low-error probability region, highlighting the limitations of shorter interleavers in achieving reliable transmission.

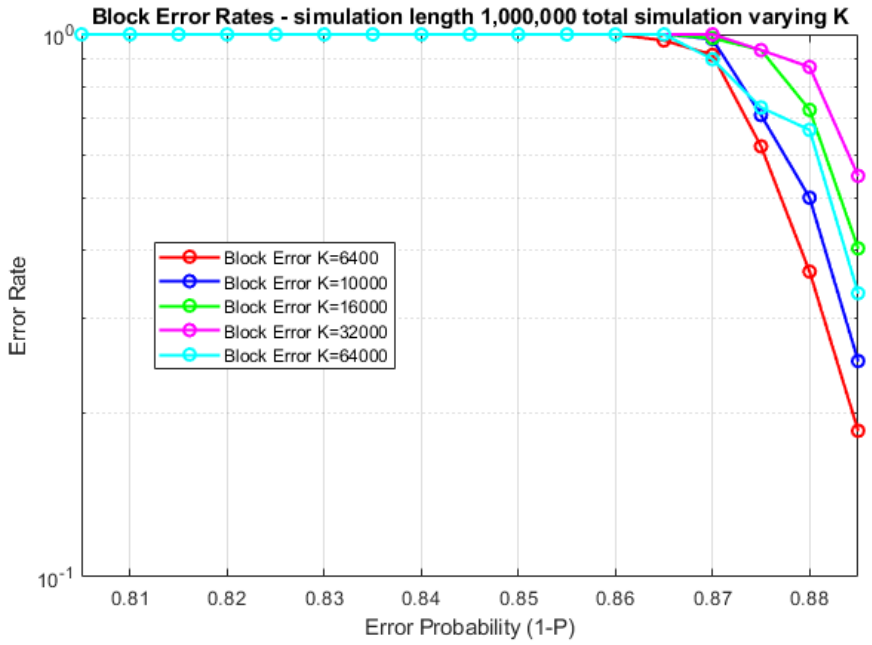


Figure 6.7: Block Error Rate (BLER) for Turbo Codes with different interleaver lengths K as a function of $1 - P$.

The figure 6.8 presents a comparative analysis of error rates in a Turbo coding scheme with a total codeword length of 1,000,000 and a block length of $K = 64{,}000$. The bit error rate (BER) and block error rate (BLER) are shown alongside theoretical estimates derived from the probability of an uncoded block error. The theoretical block error probability PBL is computed using the formula:

$$P_{BL} = 1 - (1 - p_B)^K, \tag{6.1}$$

where p_B denotes the probability of a single bit error, and K represents the block length. The results indicate that the uncoded error probability remains relatively high, while the application of Turbo coding significantly reduces both BER and BLER, especially at higher values of $(1 - P_{\text{error}})$. Notably, the simulated block error rate aligns well with the theoretical predictions at lower error probabilities but diverges as the coding gain improves. This divergence suggests the impact of interleaving and iterative decoding in practical Turbo-coded systems, which enhance error correction beyond theoretical uncoded estimations.

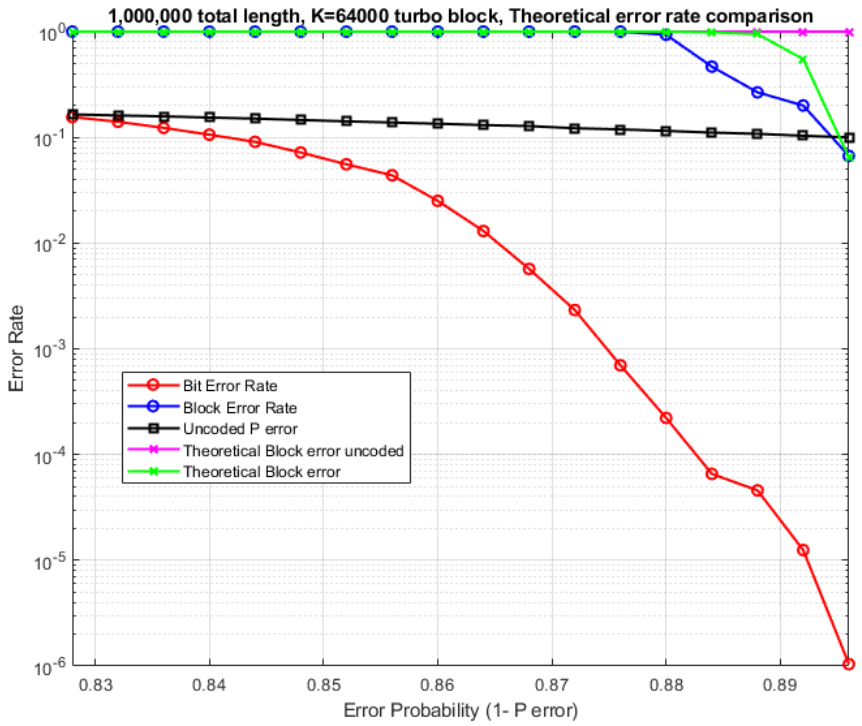


Figure 6.8: Comparison of theoretical and simulated error rates in a Turbo-coded system.

Chapter 7

Conclusions

The results presented in the previous chapter are consistent with the existing literature on turbo codes theory and CRYSTALS-Kyber software implementations. They support the ultimate goal of this project: developing a software framework that facilitates the adoption of Rust in harsh environments, enhancing security and reliability by leveraging the potential of emerging technologies.

There are numerous avenues for further exploration and testing, ranging from the utilization of specialized hardware architectures to the deployment of purpose-built computing units and real satellite communication protocols. Future research should also focus on validating these approaches through rigorous testing under strict operational constraints, ensuring robustness and resilience in space applications.

This work has demonstrated the feasibility of employing Rust for secure and reliable cryptographic and error-correcting code implementations in space systems. By integrating state-of-the-art post-quantum cryptography with high-performance error correction techniques, the work presented here lays the groundwork for subsequent studies that will further optimize these methods, bridging the gap between theoretical developments and their practical deployment in next-generation aerospace technologies.

Appendix A

Code Structure

The implementation of the proposed scheme is structured in a modular and hierarchical manner to ensure clarity, maintainability, and extensibility. The project is organized into multiple directories, each encapsulating a specific set of functionalities.

The `src` directory contains the main implementation files, structured as follows:

- **`functions`**: Includes fundamental operations such as:
  - `compress.rs` - Implements data compression techniques.
  - `encode.rs` - Handles encoding functionalities.
  - `hash.rs` - Provides cryptographic hash functions.
  - `ntt.rs` - Implements the Number-Theoretic Transform (NTT).
  - `utils.rs` - Contains utility functions used throughout the project.
- **`kem`**: Manages the Key Encapsulation Mechanism (KEM).
  - `mod.rs` - Main module handling KEM operations.
- **`pke/mod.rs`**: Implements public-key encryption functionalities.
- **`structures`**: Defines algebraic and mathematical structures.
  - `algebraics/mod.rs` - Centralizes algebraic operations.
  - `matrix.rs` - Implements matrix operations.
  - `polynomial.rs` - Provides polynomial arithmetic.
  - `polyvec.rs` - Handles polynomial vector computations.
  - `bytearray.rs` - Defines a custom byte array structure.
  - `primefield.rs` - Implements operations over finite fields.

- **`turbof`**: Implements Turbo Codes for error correction.
  - `mod.rs` - Main module for Turbo Codes.
  - `bsc_channel.rs` - Models the Binary Symmetric Channel (BSC).
  - `mapints.rs` - Contains mapping and interleaving functions.
  - `rsc_encoder.rs` - Implements Recursive Systematic Convolutional encoding.
  - `siso_decoder.rs` - Implements Soft-Input Soft-Output decoding.
  - `turbo_decoder.rs` - Implements Turbo decoding.
  - `turbo_encoder.rs` - Implements Turbo encoding.
  - `turbo_simulation.rs` - Provides a framework for Turbo Code simulations.
  - `utils.rs` - Contains additional helper functions.

Additionally, the project contains several configuration and data files:

- `input.txt` - Input data for simulations.
- `interleaver.txt` - Interleaver parameters.
- `kyber_keys.txt` - Precomputed cryptographic keys.
- `output.txt` - Stores the results of executed tests.
- `Cargo.toml` - Rust package configuration file.
- `Cargo.lock` - Dependency lock file.
- `LICENSE` - License file of the project.

The project follows best practices in Rust development, leveraging traits and generics to promote flexibility and efficiency. The modular design enhances code reusability, simplifies debugging and testing, and allows for easy extension of functionalities. Integration tests are included in the `tests` directory to validate the correctness and performance of the implementation.

Acknowledgements

I would like to thank my supervisors, Prof. Marina and Prof. Fred, for guiding me through this project, for all the good coffee, and for supporting my artistic aspirations. I am also grateful to Prof. Seth for letting me play his guitar in the office. A big thank you to all the wonderful people I met along this journey on and off stage: Tia, Jacopo, Barbara, Troy, Amar, Don, Dane, Mort, Lucas, Luz and many others.
Until we meet again.
Grazie a Marco, Michele, Francesca, Giulia, Paola, Giorgio, Samuele, Andrea, Arianna, Giulia, Jacopo, Aurora, Riccardo, Lorenzo, Andrea, Francesco. Grazie ai Cinemini e tutte le persone che sono diventate una seconda famiglia. Grazie a Serena, Stefano, Luciano, per avermi accolto nella scena torinese quando ancora non avevo vinto Martina Franca. Grazie ad Angelo, Andrea e tutte le persone con cui ho condiviso parte di questa esperienza. Grazie di cuore alla mia famiglia, ai miei genitori: tutti i traguardi che ho raggiunto sono vostri, e tutti quelli che raggiungerò.

Bibliography

[1] Atif Farid Mohammad, Pamela Almeida, Yasmin Soliman, Ajay Sadhu, Keerthi Kata, and Jeremy Straub. «Secure Satellite Database Transmission». In: *2019 IEEE Aerospace Conference*. 2019, p. 1. DOI: 10.1109/AERO.2019.8741992 (cit. on p. 2).

[2] J.T. Wallmark and S.M. Marcus. «Minimum size and maximum packaging density of non-redundant semiconductor devices». In: *Proceedings of the IRE* 50 (1962), pp. 286–298. DOI: 10.1109/AERO.2019.8741992 (cit. on p. 2).

[3] Jan Budroweit, Mattis Paul Jaksch, and Maciej Sznajder. «Proton Induced Single Event Effect Characterization on a Highly Integrated RF-Transceiver». In: *Avionic Systems* (2019) (cit. on p. 4).

[4] Federico Faccio. *Radiation effects in devices and technologies*. Lecture notes on radiation effects at CERN. 2005 (cit. on p. 5).

[5] Kul Bhasin and Jeffrey Hayden. «Developing Architectures and Technologies for an Evolvable NASA Space Communication Infrastructure». In: *NASA/TM—2004-213108* (2004) (cit. on p. 5).

[6] Michael A. Nielsen and Isaac L. Chuang. *Quantum Computation and Quantum Information*. Cambridge University Press, 2010 (cit. on p. 7).

[7] *The Rust Standard Library* (cit. on pp. 11, 12).

[8] Matt Miller. «BlueHatIL». In: *Trends, challenge, and shifts in software vulnerability mitigation.* 2019 (cit. on p. 13).

[9] IBM. *Cost of a Data Breach Report 2024*. Tech. rep. IBM, 2024 (cit. on p. 12).

[10] ONCD. «Back to the Building Blocks: A Path Toward Secure and Measurable Software». In: *ONCD* (2024) (cit. on p. 19).

[11] Lukas Seidel and Julian Beier. «Bringing Rust to Safety-Critical Systems in Space». In: *IEEE* (2024) (cit. on p. 19).

[12] ESA and DLR. *cRustacea in Space - Co-operative RUST and C embedded applications in Space - Theory and Practice*. Tech. rep. ESA, 2024 (cit. on p. 21).

[13] Joseph H. Silverman. *An Introduction to Lattices, Lattice Reduction, and Lattice-Based Cryptography*. Department of Mathematics, Box 1917, Brown University, Providence, RI 02912 USA, 1997 (cit. on p. 23).

[14] Daniele Micciancio. «Minkowski's Theorem». In: *CSE* (2014) (cit. on p. 24).

[15] João Ribeiro. «Notes 2: Cryptography from LWE and SIS». In: *FCT-UNL* (2023) (cit. on p. 27).

[16] Vadim Lyubashevsky. *Basic Lattice Cryptography The concepts behind Kyber(ML-KEM) and Dilithium (ML-DSA)*. IBM Research Europe, Zurich, 2025 (cit. on p. 29).

[17] João Ribeiro. «Notes 3 & 4: Hardness of SIS, LWE, and lattice problems». In: *FCT-UNL* (2024) (cit. on p. 33).

[18] *CRYSTALS-Kyber Algorithm Specifications And Supporting Documentation* (cit. on pp. 34, 40).

[19] Ashraf I. Mahroos Moataz M. Salah Salah S. El-Agooz. «Illuminating the Mechanism of Iterative Turbo Code Decoding Process». In: *12-th International Conference on Aerospace Sciences & Aviation Technology* (2007) (cit. on pp. 45, 58).

[20] Juntao Ni. «Turbo Codes in Satellite Communication». In: *Electronic Information Engineering Beihang University* (2012) (cit. on p. 45).

[21] Jakub Sedy, Pavel Silhavy, Ondrej Krajsa, and Ondrej Hrouza. «PERFORMANCE ANALYSIS OF TURBO CODES». In: *Communications - Scientific letters of the University of Zilina* (2013) (cit. on p. 46).

[22] Marina Mondin Fred Daneshgaran. «Design of Interleavers for Turbo Codes: Iterative Interleaver Growth Algorithms of Polynomial Complexity». In: *IEEE TRANSACTIONS ON INFORMATION THEORY, VOL. 45, NO. 6,* (1999) (cit. on pp. 53, 54).

[23] Silvio A. Abrantes. «From BCJR to turbo decoding: MAP algorithms made easier». In: *Information and Telecommunication Technology Center (ITTC) of the University of Kansas* (2004) (cit. on p. 60).