Implementación de algoritmos de criptografía post-cuántica basada en RISC-V
Implementation of post-quantum cryptography algorithms based on RISC-V

Trabajo de Fin de Grado
Curso 2024–2025

Autor
Pablo Navarro Cebrián

Director
José Luis Imaña Pascual
Luis Piñuel Moreno

Doble grado en Ingeniería Informática y Matemáticas
Facultad de Informática
Universidad Complutense de Madrid

Implementación de algoritmos de criptografía post-cuántica basada en RISC-V
Implementation of post-quantum cryptography algorithms based on RISC-V

Trabajo de Fin de Grado en Ingeniería Informática

Autor
Pablo Navarro Cebrián

Director
José Luis Imaña Pascual
Luis Piñuel Moreno

Convocatoria: *Junio 2025*
Calificación: *10 (SB)*

Doble grado en Ingeniería Informática y Matemáticas
Facultad de Informática
Universidad Complutense de Madrid

16 de junio de 2025

Dedicatoria

A mis padres, Alba y Marta

Agradecimientos

A mis padres, por confiar en mí y empujarme a sacar siempre mi mejor versión, y a mi hermana, Alba, por quererme tanto y ser mi referente. Gracias por enseñarme a luchar y no rendirme nunca.

A mi familia: abuelas, abuelos, tías, tíos y, por supuesto, mis primos. Os quiero con locura a todos. En especial, quiero agradecer a mi primo Rodrigo: un ejemplo a seguir y a quien admiro profundamente. Algún día quiero ser como tú.

A mis amigos de la carrera, por haber hecho de estos cinco años los mejores de mi vida. Sin vosotros, nada habría sido igual.

A Martita, por ser mi apoyo incondicional y la persona que me hace sonreír cada día. Gracias por apoyarme en los momentos más difíciles y celebrar conmigo cada logro como si fuera tuyo. Te quiero.

Resumen

Implementación de algoritmos de criptografía post-cuántica basada en RISC-V

El rápido avance de la computación cuántica amenaza la seguridad de los esquemas criptográficos clásicos, incluidos métodos ampliamente utilizados como RSA y ECC. Como respuesta ha surgido la criptografía post-cuántica (PQC), siendo los esquemas basados en retículos, como Kyber, los principales candidatos debido a sus sólidas garantías de seguridad. Este proyecto se centra en la implementación eficiente y optimización de Kyber en arquitecturas RISC-V, explorando tanto la versión de referencia como una versión optimizada del algoritmo. Se presta especial atención a la multiplicación modular, donde se integra la aritmética de Plantard y se evalúa frente a la multiplicación de Montgomery estándar. El estudio se lleva a cabo en dos plataformas RISC-V, las placas K230 y Banana Pi F3, cada una con características microarquitectónicas distintas. Los resultados proporcionan información sobre los compromisos de rendimiento entre las diferentes técnicas de multiplicación modular y demuestran la viabilidad de implementar esquemas de PQC en plataformas de hardware abiertas y flexibles como RISC-V.

Palabras clave

Criptografía Post-Cuántica (PQC), RISC-V, Criptografía Basada en Retículos, Kyber, Multiplicación Modular, Aritmética de Plantard, Estandarización NIST.

Abstract

Implementation of post-quantum cryptography algorithms based on RISC-V

The rapid advancement of quantum computing threatens the security of classical cryptographic schemes, including widely used methods such as RSA and ECC. As a response, post-quantum cryptography (PQC) has emerged, with lattice-based schemes like Kyber becoming leading candidates due to their strong security guarantees. This project focuses on the efficient implementation and optimization of Kyber on RISC-V architectures, exploring both the reference and an optimized version of the algorithm. Special attention is given to modular multiplication, where Plantard arithmetic is integrated and evaluated against the standard Montgomery multiplication. The study is conducted across two RISC-V platforms, the K230 and Banana Pi F3 boards, each with distinct microarchitectural characteristics. The results provide insights into the performance trade-offs of different modular multiplication techniques and demonstrate the feasibility of deploying PQC schemes on open, flexible hardware platforms like RISC-V.

Keywords

Post-Quantum Cryptography (PQC), RISC-V, Lattice-Based Cryptography, Kyber, Modular Multiplication, Plantard Arithmetic, NIST Standardization.

Contents

List of tables

Chapter 1

Introduction

1.1. Motivation

With the advancement of quantum computing, traditional public-key cryptosystems such as RSA and ECC are becoming increasingly vulnerable. Post-quantum cryptography (PQC) aims to develop cryptographic schemes that are secure against quantum adversaries. Among the various PQC proposals, lattice-based schemes like Kyber have emerged as strong candidates due to their balance of security and efficiency.

As PQC schemes are expected to be deployed across a wide range of devices, from servers to constrained embedded systems, optimizing their performance becomes critical. In this context, the RISC-V architecture presents a promising platform due to its open standard, modularity and increasing adoption. This research explores the optimization of post-quantum cryptography on RISC-V hardware to better understand the trade-offs and opportunities in such environments.

1.2. Objectives of the Project

The main objective of this project is to evaluate the performance of the Kyber post-quantum key encapsulation mechanism on RISC-V architectures by benchmarking and analysing different implementations. Kyber is one of the leading candidates selected by NIST for post-quantum cryptographic standardization, making its efficient deployment on various hardware platforms a critical area of research.

This project specifically aims to:

- Benchmark both the reference and optimized versions of Kyber across two RISC-V platforms with different performance profiles and micro-architectural features.
- Implement and evaluate an alternative approach to modular multiplication based on Plantard arithmetic, with the goal of improving performance in key components such as polynomial operations.

- Analyse the performance trade-offs between the standard and alternative approaches to modular arithmetic on RISC-V.

1.3. Methodology and Work Plan

The project was carried out following a progressive and structured approach, starting from foundational learning and culminating in implementation, benchmarking and analysis. The main steps followed are described below:

1. **Initial Background Study:** Given my initial lack of knowledge in cryptography, the first step was to read a book on classical cryptography to establish the necessary theoretical foundation. This allowed me to understand core cryptographic principles and the motivations behind secure communication.

2. **Post-Quantum Cryptography Research:** Once familiar with classical cryptography, I focused on post-quantum cryptography (PQC), paying special attention to lattice-based cryptography. I studied the Learning With Errors (LWE) problem and its ring-based variants, which underpin the security of many PQC schemes. After exploring the set of algorithms standardized by NIST, I decided to focus my project on Kyber. This choice was guided by two factors: its strong mathematical foundation based on Module-LWE, which aligned with my academic background in Mathematics, and its efficiency, which makes it well-suited for constrained environments such as those offered by embedded RISC-V platforms. These attributes made Kyber an ideal candidate for both theoretical study and practical implementation.

3. **Kyber Analysis:** With the required theoretical background, I studied the Kyber Key Encapsulation Mechanism in detail to understand its algorithmic structure. This deep understanding was necessary in order to implement modifications later on, particularly the substitution of Montgomery multiplication with Plantard arithmetic in the modular multiplication operations.

4. **RISC-V Learning Phase:** In parallel, I began learning about the RISC-V architecture, including its instruction set, memory model and performance characteristics.

5. **Plantard Arithmetic Implementation:** I read academic resources and documentation on Plantard arithmetic, a less common but potentially efficient technique for modular multiplication. I implemented two variants of this method and prepared them for integration into the Kyber reference implementation.

6. **Benchmarking Strategy:** To perform consistent and meaningful comparisons, I evaluated various benchmarking methods and chose to use `perf`, as it provides detailed and comparable performance metrics across different platforms.

7. **Optimized Implementation Study:** I studied the optimized version of Kyber to understand the techniques used to enhance its performance, such as efficient polynomial arithmetic and vectorization.

8. **Integration and Benchmarking:** The custom Plantard-based modular multiplication routines were integrated into the Kyber reference implementation. I conducted extensive benchmarks, including:

   - Performance comparisons across different RISC-V boards.
   - Evaluation of the reference versus the optimized implementation.
   - Analysis of the impact of using Plantard arithmetic.

9. **Result Analysis:** The final step involved critically analysing the collected data to draw conclusions regarding the effectiveness of optimization strategies, hardware-software interactions and the potential of Plantard arithmetic for PQC applications.

1.4. Structure of the Document

This document is structured into six main chapters and an appendix, aiming to guide the reader progressively from background knowledge to the final contributions and results of the project.

- **Chapter 1: Introduction** outlines the motivation behind the project, its objectives and the methodology followed.
- **Chapter 2: State of the Art** presents a detailed review of classical and quantum cryptography, the impact of quantum computing and the emergence of post-quantum cryptographic schemes such as Kyber.
- **Chapter 3: Theoretical Foundations** introduces the mathematical problems underlying lattice-based cryptography, explains the Kyber algorithm in depth and describes the RISC-V architecture and modular multiplication techniques relevant to the project.
- **Chapter 4: Development and Optimization Proposal** describes the hardware platforms used, the methodology for benchmarking and the details of the integration and optimization efforts carried out during the project.
- **Chapter 5: Results and Discussion** presents the outcomes of the benchmarking across different platforms, compares the reference and optimized implementations, evaluates the use of Plantard modular multiplication and critically analyses the results.
- **Chapter 6: Conclusions and Future Work** summarizes the main achievements of the project, discusses its applicability and proposes directions for future research and improvements.

- **Appendix A** includes additional technical details, namely the code used for generating zeta values and the corresponding output.

Chapter 2

State of the art

2.1. Classical Cryptography: Principles and Security Foundations

Cryptography is the foundation of secure communication, ensuring confidentiality, integrity and authenticity of data. Classical cryptographic techniques are broadly categorized into symmetric and asymmetric cryptography, each with distinct principles and use cases. (Buchmann, 2004)

2.1.1. Symmetric Cryptography

Symmetric cryptography, also known as secret-key cryptography, relies on a single shared key for both encryption and decryption. This approach is computationally efficient and widely used in applications requiring high-speed encryption, such as secure communication channels and data storage.

One of the most well-known symmetric encryption standards is the Advanced Encryption Standard (AES), which has become the global standard due to its strong security properties and efficiency. Symmetric cryptography ensures data protection under the assumption that the shared key remains secret between the communicating parties. However, key distribution poses a significant challenge, leading to the development of asymmetric cryptographic methods.

2.1.2. Asymmetric Cryptography

Asymmetric cryptography, or public-key cryptography, uses a pair of mathematically related keys: a public key for encryption and a private key for decryption. This eliminates the need for a shared secret key and facilitates secure communication over untrusted networks.

RSA (Rivest-Shamir-Adleman) and Elliptic Curve Cryptography (ECC) are two of the most commonly used asymmetric cryptographic schemes. Their security is based on computationally hard problems: RSA relies on the difficulty of integer factorization while ECC depends on the hardness of the elliptic curve discrete logarithm

problem. Asymmetric cryptography is widely employed for secure key exchange, digital signatures and authentication mechanisms.

2.1.3. Hash Functions

Cryptographic hash functions play a crucial role in data integrity verification and digital signatures. A hash function takes an arbitrary-length input and produces a fixed-length output, known as a hash or digest, which uniquely represents the original data. Even a small change in the input results in a drastically different output, making hash functions highly effective for integrity checking.

Hash functions are widely used in digital signatures, where they provide a compact representation of a message before it is signed with a private key. This ensures both efficiency and security, as the signature is applied to a fixed-length digest rather than the entire message. Additionally, hash functions enable password verification, file integrity checking and blockchain security mechanisms.

2.1.4. Security Assumptions

The security of classical cryptographic schemes depends on assumptions regarding the computational hardness of specific mathematical problems. The integer factorization problem and discrete logarithm problem underpin the robustness of different cryptographic protocols. As long as these problems remain infeasible to solve with classical computational methods, cryptographic schemes built on them remain secure.

While classical cryptography has effectively secured digital communications for decades, the emergence of new computational paradigms challenges its long-term viability. In particular, novel computing models have the potential to undermine traditional security assumptions, making the exploration of alternative cryptographic approaches necessary, as discussed in the next section.

2.2. Quantum Computing: Capabilities and Cryptographic Implications

Quantum computing represents a paradigm shift in computational theory and practice, leveraging the principles of quantum mechanics to process information in fundamentally different ways compared to classical computers. Unlike classical bits, which exist in one of two states (0 or 1), quantum bits (qubits) can exist in a superposition of both states simultaneously. This property, combined with entanglement and quantum interference, enables quantum computers to solve certain problems exponentially faster than their classical counterparts. (Mavroeidis et al., 2018)

2.2.1. Fundamental Principles of Quantum Computing

Qubits possess three key properties that enable their computational advantage:

- **Superposition**: A qubit can exist in a linear combination of the states $|0\rangle$ and $|1\rangle$, meaning it can represent multiple possible states at once. This allows quantum computers to perform multiple calculations in parallel.
- **Entanglement**: When two qubits become entangled, their states are correlated regardless of the distance between them. This phenomenon enables efficient information transfer and computation beyond classical capabilities.
- **Quantum Interference**: The probability amplitudes of quantum states can interfere with each other constructively or destructively, guiding a quantum algorithm toward the correct solution by amplifying desired outcomes and cancelling out incorrect ones.

2.2.2. Differences Between Quantum and Classical Computing

While classical computers execute algorithms through deterministic state transitions, quantum computers leverage probabilistic computation. The key distinctions include:

- **Parallelism**: Classical computers must evaluate each possible solution sequentially or in parallel using multiple processors, whereas quantum computers exploit superposition to evaluate many solutions simultaneously.
- **Algorithmic Speedups**: Certain quantum algorithms, such as those for factoring large numbers or searching unsorted databases, outperform the best-known classical algorithms by significant margins.
- **Hardware Constraints**: Quantum systems require highly controlled environments, often operating at near absolute zero temperatures to maintain qubit coherence, whereas classical systems function under more flexible conditions.

2.2.3. Cryptographic Implications

Quantum computing poses a direct threat to modern cryptographic systems, particularly those based on the difficulty of factoring large numbers or computing discrete logarithms. The most widely used public-key cryptosystems, such as RSA, ECC and Diffie-Hellman, rely on these hard problems for security. A sufficiently powerful quantum computer would render them obsolete by executing quantum algorithms that efficiently solve these problems. Furthermore, symmetric cryptographic schemes also face potential risks. While quantum computers do not outright break symmetric encryption, they reduce its effective security level by enabling faster key search attacks, necessitating a shift toward larger key sizes for continued security.

Given these challenges, post-quantum cryptography has emerged as an essential field, focusing on developing cryptographic primitives that remain secure even in the presence of quantum adversaries. These include lattice-based, hash-based and code-based cryptographic constructions (Bernstein et al., 2009).

In the next section, we delve into two of the most impactful quantum algorithms: Shor's algorithm, which threatens the security of widely used public-key cryptosystems, and Grover's algorithm, which affects symmetric key cryptography and hash functions.

2.3. Shor's and Grover's Algorithms

Quantum computing introduces a new computational paradigm that challenges classical cryptographic assumptions. Two of the most significant quantum algorithms in this context are **Shor's algorithm** and **Grover's algorithm**, both of which threaten the security of widely used cryptographic schemes.

2.3.1. Shor's Algorithm: Breaking Public-Key Cryptography

Shor's algorithm, developed by Peter Shor in 1994, proves that quantum computers can efficiently solve problems that are computationally infeasible for classical computers. Specifically, it can factor large integers and compute discrete logarithms in polynomial time, which are two problems that form the foundation of many public-key cryptosystems. (Shor, 1997)

- **Impact on RSA**: The security of RSA relies on the assumption that factoring a large composite number into its prime components is computationally difficult. Classical algorithms for integer factorization, such as the General Number Field Sieve, require superpolynomial time, making RSA secure under current computing models. However, Shor's algorithm can factor large numbers exponentially faster, rendering RSA insecure once sufficiently powerful quantum computers become available.

- **Impact on ECC and Diffie-Hellman**: Elliptic Curve Cryptography (ECC) and the Diffie-Hellman key exchange depend on the hardness of the discrete logarithm problem over finite fields or elliptic curves. Just as Shor's algorithm efficiently factors large integers, it also solves the discrete logarithm problem in polynomial time, completely breaking these cryptographic schemes.

In summary, Shor's algorithm poses an existential threat to most asymmetric cryptosystems currently used for secure communication, digital signatures and key exchange. Once large-scale quantum computers become practical, these cryptosystems will no longer provide security, making it essential to develop alternatives resistant to quantum attacks.

2.3.2. Grover's Algorithm: Weakening Symmetric Cryptography

Unlike Shor's algorithm, which directly breaks asymmetric cryptography, Grover's algorithm, introduced by Lov Grover in 1996, affects symmetric cryptographic schemes by providing a quantum speed-up for brute-force search problems (Grover, 1996).

- **Impact on Symmetric Encryption (e.g., AES)**: Classical brute-force attacks on symmetric encryption require checking all possible keys, making security dependent on key length. Grover's algorithm can perform an exhaustive search in quadratic time, meaning that an n-bit key provides only $\frac{n}{2}$ bits of quantum security. For example, AES-128, which is classically secure against brute-force attacks, would only offer an effective security level of 64 bits against quantum attacks, making it vulnerable. To maintain its original 128-bit security level in a post-quantum scenario, it would be necessary to adopt AES-256 instead.

- **Impact on Hash Functions**: Hash functions rely on preimage and collision resistance. A classical brute-force attack on a hash function with an m-bit output requires 2^m operations. Grover's algorithm reduces this complexity to $2^{\frac{m}{2}}$, significantly weakening the strength of cryptographic hash functions like SHA-2 and SHA-3. To maintain equivalent security levels, cryptographic standards may require increasing hash output sizes, such as moving from SHA-256 to SHA-512.

Although Grover's algorithm does not completely break symmetric cryptography as Shor's algorithm does with asymmetric cryptography, it forces adjustments to be made in key sizes and hash function parameters to maintain security in a quantum world.

2.4. Post-Quantum Cryptography: A Future Necessity

As discussed in the previous sections, the rise of quantum computing threatens the security of widely used cryptographic protocols. Shor's algorithm enables the efficient factorization of large numbers and the computation of discrete logarithms, rendering RSA, ECC and Diffie-Hellman insecure. Grover's algorithm, while less disruptive, weakens the security of symmetric cryptography by reducing the effective key size. This growing vulnerability has driven researchers to explore new cryptographic schemes that can withstand quantum attacks, an area known as **post-quantum cryptography (PQC)**.

Post-quantum cryptography seeks to develop cryptographic algorithms that remain secure even against adversaries equipped with powerful quantum computers. Unlike quantum cryptography, which leverages quantum mechanics to establish secure communication, PQC operates within classical computing frameworks, ensuring

compatibility with existing digital infrastructure. The basic types of post-quantum cryptographic algorithms include (Bernstein et al., 2009):

- **Lattice-based cryptography**: Relies on the hardness of problems such as Learning With Errors (LWE) and Shortest Vector Problem (SVP). This approach is among the most promising due to its strong security proofs and efficient implementations.
- **Code-based cryptography**: Based on the difficulty of decoding random linear codes, as used in schemes like McEliece.
- **Multivariate-quadratic cryptography**: Relies on the difficulty of solving systems of multivariate quadratic equations.
- **Hash-based signatures**: Constructed from cryptographic hash functions, offering secure digital signatures resistant to quantum attacks.

Among these approaches, lattice-based cryptography has emerged as a strong candidate due to its balance of security and efficiency, making it a focal point of standardization efforts.

2.4.1. NIST Standardization and the Role of Kyber

Recognizing the urgency of transitioning to quantum-resistant cryptographic systems, the *National Institute of Standards and Technology* (NIST) launched a multiphase standardization process in 2016 to evaluate and select PQC algorithms for widespread adoption. The competition aimed to identify algorithms suitable for public-key encryption, key exchange and digital signatures.

After multiple evaluation rounds, **Kyber**, a lattice-based key encapsulation mechanism (KEM), was selected as the primary standard for post-quantum encryption. Kyber's security is rooted in the Learning With Errors (LWE) problem, which is thought to be computationally infeasible even for quantum computers. In addition to its strong security foundations, Kyber offers efficient performance, making it suitable for real-world applications.

The adoption of post-quantum cryptographic standards marks a critical step towards securing digital infrastructure against future quantum threats. As organizations and governments begin the transition, continued research and optimization will be necessary to ensure the seamless integration of quantum-resistant algorithms into modern cryptographic systems.

Chapter 3

Theoretical foundations

3.1. Underlying Mathematical Problems: LWE and Lattices

Modern cryptographic security is deeply rooted in the hardness of mathematical problems that remain intractable even for quantum computers. One of the most significant of these problems is the Learning With Errors (LWE) problem, which has become the foundation for many post-quantum cryptographic schemes, including Kyber. The strength of LWE-based cryptosystems lies in their connection to lattice problems, which have been extensively studied in computational mathematics and are widely regarded as computationally difficult.

This section provides an overview of the LWE problem, its relation to hard lattice problems and the key theorems that prove the reduction of worst-case lattice problems to LWE. Additionally, we introduce Module-LWE, a structured variant that improves efficiency while maintaining strong security guarantees.

3.1.1. The Learning With Errors problem (LWE)

The LWE problem was introduced by Oded Regev (2005) as a versatile mathematical problem that is both theoretically interesting and practically useful for cryptographic applications. At a high level, LWE can be thought of as a problem of solving noisy linear equations, where the presence of small errors makes the system hard to solve. (Regev, 2009)

More formally, the problem asks to recover a secret $\mathbf{s} \in \mathbb{Z}_q^n$ given a sequence of approximate random linear equations of the form:

$$a_i^1 s_1 + a_i^2 s_2 + ... + a_i^n s_n \approx b_i \ (mod\ q)$$

where:

- The vectors $\mathbf{a_i}$ are chosen uniformly at random from $\mathbb{Z}_q^n$.

- The output value is computed as $b_i = \langle \mathbf{a_i}, \mathbf{s} \rangle + e_i$, where the error term e_i follows a probability distribution on $\mathbb{Z}_q$ (typically a discrete Gaussian with small standard deviation).

If no error terms e_i were present, the problem would be trivial, as it could be solved efficiently using Gaussian elimination. However, the introduction of noise significantly increases its complexity. Specifically:

- Any attempt to solve LWE using direct linear algebra will amplify the noise, making the equations unusable.
- The problem becomes analogous to decoding from random linear codes or solving bounded distance decoding (BDD) problems on lattices, both of which are known to be computationally difficult.

3.1.2. The Hardness of LWE and its Connection to Lattice Problems

Lattices are mathematical structures that consist of discrete points in multi-dimensional space arranged in a regular pattern. Many well-known computational problems on lattices, such as the Shortest Vector Problem (SVP) and the Bounded Distance Decoding (BDD) problem, are believed to be computationally intractable even for quantum computers.

The hardness of LWE is supported by reductions that show solving LWE is at least as hard as solving worst-case lattice problems. These reductions provide a theoretical foundation for the security of LWE-based cryptosystems (Regev, 2010). Avoiding some technical details regarding the restrictions of the modulus q, the error distribution and so on, three fundamental theorems establish this connection:

1. **Search-LWE to BDD Reduction**: If there exists an algorithm that can recover the secret $\mathbf{s}$ from a set of random linear noisy equations (the search version of LWE), then there exists an efficient quantum algorithm that solves the Bounded Distance Decoding (BDD) problem, meaning it can recover the closest lattice point to a given point that is close to the lattice.
2. **Decision to Search Reduction**: If there exists an algorithm that can distinguish LWE samples from uniformly random samples (the decision version of LWE) with probability exponentially close to 1, then there exists an efficient algorithm that solves the search version of LWE, meaning it can recover the secret $\mathbf{s}$ with probability exponentially close to 1.
3. **Average-case to Worst-case Reduction**: If there exists an algorithm that can distinguish LWE samples from uniformly random samples with probability exponentially close to 1 for a non-negligible fraction of all possible $\mathbf{s}$ (Average-case), then there exists an efficient algorithm that solves the decision version of LWE, meaning it can distinguish LWE samples from uniformly random samples with probability exponentially close to 1 for all $\mathbf{s}$ (Worst-case).

To summarize these three fundamental theorems that establish the connection between the hardness of LWE and lattice problems, we can conclude that the ability to distinguish between average LWE samples and uniformly random samples implies the existence of an efficient quantum solution to worst-case lattice problems, such as the Bounded Distance Decoding (BDD) problem. Since these lattice problems are believed to be computationally intractable, even for quantum computers, this indicates that solving worst-case lattice problems is as hard as solving the average-case decision version of LWE. This makes LWE a strong candidate for quantum-resistant cryptography, as its hardness is rooted in problems that remain challenging even in the presence of quantum algorithms.

The efficiency of LWE-based cryptosystems is limited by the large key and ciphertext sizes that are required for security. Specifically, the public key size in an LWE-based system grows as $\mathcal{O}(mn \log q) = \tilde{\mathcal{O}}(n^2)$, where n is the security parameter, m the number of equations and q is the modulus, making it impractical for large-scale applications. Additionally, encryption in the LWE scheme increases the message size by a factor of $\mathcal{O}(n \log q) = \tilde{\mathcal{O}}(n)$, which results in significant overhead.

3.1.3. Ring-LWE and Module-LWE: More Efficient Variants

While LWE provides strong security guarantees, its direct use in cryptographic systems is impractical due to large key sizes and slow computations. To address these issues, structured variants like Ring-LWE (RLWE) and Module-LWE (MLWE) were introduced. These variants improve efficiency while preserving security, making them suitable for real-world applications. (Wang and Wang, 2019)

3.1.3.1. Ring-LWE: Leveraging Polynomial Rings for Efficiency

Ring-LWE modifies LWE by replacing large vectors with polynomials over a ring, reducing storage and computational costs. Instead of working with matrices and vectors, RLWE operates in the polynomial ring $R_q = \mathbb{Z}_q[x]/(f(x))$, where $f(x)$ is a carefully chosen polynomial, often a cyclotomic polynomial. This structure provides several advantages:

- **Smaller key sizes**: RLWE significantly reduces public and private key sizes compared to standard LWE ($\mathcal{O}(n)$ versus $\mathcal{O}(n^2)$).
- **Faster computations**: Operations such as encryption and decryption involve polynomial arithmetic, which can be efficiently implemented using the Number Theoretic Transform (NTT), reducing computational overhead.

3.1.3.2. Module-LWE: A Practical Compromise

While RLWE improves efficiency, it introduces additional algebraic structure that could potentially be exploited by attackers. To balance efficiency and security, MLWE was introduced, which generalizes LWE while keeping some advantages of RLWE. Instead of working with single polynomials (RLWE) or large vectors (LWE),

MLWE organizes data into small matrices of polynomials, allowing better parameter tuning and flexibility. This scheme provides the following advantages:

- **Improved Efficiency**: MLWE organizes data into polynomial matrices, allowing for more efficient computation compared to LWE's large vectors. This leads to faster encryption and decryption.
- **Smaller Key Sizes**: MLWE enables smaller key sizes while retaining strong security guarantees, making it more suitable for practical applications where large key sizes in LWE would be inefficient.
- **Stronger Security**: MLWE's structure avoids some of the potential vulnerabilities of Ring-LWE. Its matrix-based approach reduces the risk of certain attacks, ensuring that security remains robust even with quantum computing threats.

To conclude, Ring-LWE and Module-LWE offer substantial improvements over standard LWE, with MLWE emerging as the preferred choice for cryptographic protocols like Kyber. By structuring data into polynomial modules, MLWE achieves an optimal balance between security, efficiency and practicality, making it a strong foundation for modern post-quantum cryptography.

3.2. Kyber: Algorithm Description and Relevance

Kyber is a post-quantum cryptographic key-encapsulation mechanism (KEM) designed to establish secure shared secrets between parties over public channels, even in the presence of quantum computing threats. A Key Encapsulation Mechanism (KEM) is a cryptographic primitive designed for secure key exchange, particularly in asymmetric encryption schemes. Instead of encrypting arbitrary messages directly, a KEM allows one party to encapsulate a randomly generated shared secret using a recipient's public key. The recipient can then decapsulate the ciphertext using their private key to retrieve the shared secret. This shared secret can subsequently be used as a symmetric key for encrypting actual messages using efficient symmetric encryption schemes like AES.

The security of Kyber is based on the hardness of the Module Learning With Errors (MLWE) problem. Kyber has been standardized by the National Institute of Standards and Technology (NIST) as the Module-Lattice-Based Key Encapsulation Mechanism (ML-KEM) in Federal Information Processing Standards Publication 203 (FIPS 203).

3.2.1. Algorithm Overview

The encryption and decryption processes in Kyber involve structured noise, modular arithmetic and efficient polynomial arithmetic over rings to provide both security and efficiency. Kyber operates through three primary algorithms: key generation, encapsulation and decapsulation. (National Institute of Standards and Technology, 2024)

3.2.1.1. Key Generation: Establishing the Public and Private Keys

Before encryption and decryption, key generation is performed to establish the public and private keys:

1. **Generating the secret and public key**:
   - The private key consists of a small-norm polynomial vector $\mathbf{s}$ sampled from a centered binomial distribution (CBD).
   - A corresponding public key is generated using a uniformly random matrix $\mathbf{A}$ and the MLWE equation:

     $$\mathbf{t} = \mathbf{A} \cdot \mathbf{s} + \mathbf{e} \; mod \; q$$

     where $\mathbf{e}$ is an error term drawn from a CBD.

     The public key is $(\mathbf{A}, \mathbf{t})$.
2. **Compression for efficiency**: Both $\mathbf{t}$ and $\mathbf{s}$ are compressed using carefully chosen bit-width reductions to balance efficiency and security.

3.2.1.2. Encryption: Generating a Ciphertext

Encryption in Kyber follows the MLWE paradigm and ensures that the ciphertext hides the message under structured noise. Given a public key $(\mathbf{A}, \mathbf{t})$, encryption works as follows:

1. **Sampling random polynomials**:
   - A temporary secret $\mathbf{y}$ (a small-norm polynomial vector) is chosen from a CBD.
   - A small-norm error vector $\mathbf{e_1}$ and a small error polynomial e_2 are also sampled from a CBD.
2. **Computing the ciphertext components**:
   - The first component $\mathbf{u}$ of the ciphertext is calculated as:

     $$\mathbf{u} = \mathbf{A}^T \cdot \mathbf{y} + \mathbf{e_1} \; mod \; q$$

   - The second component $\mathbf{v}$ is computed as:

     $$\mathbf{v} = \mathbf{t}^T \cdot \mathbf{y} + e_2 + \mathbf{m} \cdot \lfloor \frac{q}{2} \rfloor \; mod \; q$$

     where $\mathbf{m}$ is the message to be encrypted.
3. **Ciphertext compression**: The components $\mathbf{u}$ and $\mathbf{v}$ are compressed before transmission to improve efficiency.

The ciphertext consists of $(\mathbf{u}, \mathbf{v})$, which can be sent over an insecure channel to the receiver.

3.2.1.3. Decryption: Recovering the Original Message

Decryption in Kyber relies on the fact that $\mathbf{s}$ is known to the receiver and that the MLWE structure allows the original message to be retrieved with high probability.

1. **Computing an approximation of $\mathbf{v}$**: The receiver computes:

$$\mathbf{v}' = \mathbf{s}^T \cdot \mathbf{u} \; mod \; q$$

Since $\mathbf{u} = \mathbf{A}^T \cdot \mathbf{y} + \mathbf{e_1}$:

$$\mathbf{v}' = \mathbf{s}^T \cdot (\mathbf{A}^T \cdot \mathbf{y} + \mathbf{e_1}) = \mathbf{s}^T \cdot \mathbf{A}^T \cdot \mathbf{y} + \mathbf{s}^T \cdot \mathbf{e_1} \; mod \; q$$

Moreover, we know $\mathbf{t} = \mathbf{A} \cdot \mathbf{s} + \mathbf{e}$, which implies $\mathbf{A} \cdot \mathbf{s} = \mathbf{t} - \mathbf{e}$, and, consequently:

$$\mathbf{v}' = (\mathbf{t}^T - \mathbf{e}^T) \cdot \mathbf{y} + \mathbf{s}^T \cdot \mathbf{e_1} = \mathbf{t}^T \cdot \mathbf{y} - \mathbf{e}^T \cdot \mathbf{y} + \mathbf{s}^T \cdot \mathbf{e_1} \; mod \; q$$

2. **Recovering $\mathbf{m}$**: For computing the decrypted message $\mathbf{m}'$ we first compute an approximation of $\mathbf{m}$ as:

$$\begin{aligned} \mathbf{v} - \mathbf{v}' &= (\mathbf{t}^T \cdot \mathbf{y} + e_2 + \mathbf{m} \cdot \lfloor \frac{q}{2} \rfloor) - (\mathbf{t}^T \cdot \mathbf{y} - \mathbf{e}^T \cdot \mathbf{y} + \mathbf{s}^T \cdot \mathbf{e_1}) \\ &= e_2 + \mathbf{m} \cdot \lfloor \frac{q}{2} \rfloor + \mathbf{e}^T \cdot \mathbf{y} - \mathbf{s}^T \cdot \mathbf{e_1} \\ &\approx \mathbf{m} \cdot \lfloor \frac{q}{2} \rfloor \end{aligned}$$

So:

- If $\mathbf{v} - \mathbf{v}'$ is closer to 0 than to $\lfloor \frac{q}{2} \rfloor$, then $\mathbf{m}' = 0$
- If $\mathbf{v} - \mathbf{v}'$ is closer to $\lfloor \frac{q}{2} \rfloor$ than to 0, then $\mathbf{m}' = 1$

3. **Error correction for reliability**: To ensure robustness, Kyber applies a reconciliation mechanism to correct minor decryption errors due to noise.

These steps ensure that only the intended recipient can access the shared secret, maintaining confidentiality and integrity.

3.2.2. Use of SHA-3 Functions

Kyber makes use of cryptographic hash functions from the SHA-3 family (National Institute of Standards and Technology, 2015), specifically the Keccak permutation, in several parts of its operation:

- **Randomness Generation**: SHAKE-128 and SHAKE-256, extendable output functions (XOFs) from the SHA-3 family, are employed to generate cryptographic randomness needed during key generation and encapsulation.
- **Hashing Operations**: SHA3-256 and SHA3-512 are used for hashing operations within the algorithm to ensure data integrity and security.

The use of these standardized hash functions contributes to Kyber's robustness and security.

3.2.3. Parameter Sets and Security Levels

Kyber offers three parameter sets depending on the size of the matrix of polynomials A, each providing different levels of security and performance trade-offs:

- **ML-KEM-512**: Offers the lowest security level (128-bit security), suitable for applications where performance is prioritized over maximum security.
- **ML-KEM-768**: Provides a balanced approach, offering medium security (192-bit security) with reasonable performance.
- **ML-KEM-1024**: Delivers the highest security level (256-bit security), appropriate for scenarios requiring robust protection. However, this comes with increased computational requirements.

3.2.4. Relevance and Selection for This Study

Kyber was chosen for this study due to its strong security foundations, efficiency and flexibility. Its reliance on the MLWE problem ensures resistance against quantum attacks, aligning with the need for future-proof cryptographic solutions. Additionally, Kyber's performance is favourable compared to other post-quantum algorithms, making it suitable for practical deployment. The inclusion of multiple parameter sets allows for tailored security, accommodating various application requirements. Furthermore, its standardization by NIST as ML-KEM in FIPS 203 underscores its credibility and readiness for widespread adoption.

Kyber's efficiency and flexibility make it not only a strong cryptographic choice but also a practical candidate for hardware optimization. Efficient implementations of Kyber benefit from specialized architectures that accelerate arithmetic operations, particularly modular multiplication. In this context, the **RISC-V architecture**, with its customizable instruction set, provides an ideal platform for optimizing post-quantum cryptographic schemes.

3.3. Polynomial Multiplication and the Number Theoretic Transform (NTT)

Efficient polynomial multiplication is a fundamental operation in Kyber, as it appears extensively in key generation, encryption and decryption. Since these operations rely on the arithmetic of polynomials over finite rings, their performance directly influences the overall efficiency of the scheme.

In Kyber, all polynomial computations are performed in the ring

$$R_q := \mathbb{Z}_q[X]/(X^n + 1),$$

where $\mathbb{Z}_q$ denotes the ring of integers modulo a prime $q = 3329$ and $n = 256$ is the degree of the polynomial modulus. Elements of R_q are polynomials of the form

$$f(X) = f_0 + f_1 X + \cdots + f_{255} X^{255},$$

with coefficients $f_i \in \mathbb{Z}_q$. The modulus X^n+1 enforces a reduction rule that ensures all arithmetic remains within degree $n-1$, making R_q a finite ring of polynomials.

Naively multiplying two polynomials in R_q has a time complexity of $\mathcal{O}(n^2)$, which is computationally expensive for $n = 256$ and unsuitable for high-performance cryptographic applications. To address this inefficiency, Kyber employs the Number-Theoretic Transform (NTT), a finite-field analogue of the Discrete Fourier Transform (DFT). The NTT enables multiplication to be carried out as pointwise products in the transform domain, reducing the overall complexity to $\mathcal{O}(n \log n)$.

This section introduces the mathematical structure of the NTT, its use in transforming and multiplying polynomials in Kyber and how it enables high-performance implementations suitable for post-quantum cryptography.

3.3.1. Structure of NTT and INTT in Kyber

The Number Theoretic Transform (NTT) used in Kyber is a crucial optimization that allows polynomial multiplication in the ring $R_q = \mathbb{Z}_q[X]/(X^n + 1)$ to be performed efficiently. This is achieved by transforming elements of R_q into a more suitable representation for computation (National Institute of Standards and Technology, 2024).

The NTT exploits the fact that the ring R_q is isomorphic to another ring T_q, defined as:

$$T_q := \bigoplus_{i=0}^{127} \mathbb{Z}_q[X]/(X^2 - \zeta^{2\text{BitRev}_7(i)+1}),$$

where $\zeta = 17$ is a primitive 256-th root of unity modulo q and `BitRev7(i)` denotes the bit-reversal of the 7-bit binary representation of i. This isomorphism allows the multiplication of polynomials in R_q to be mapped into a set of smaller, independent multiplications in T_q, each modulo a degree-2 polynomial.

The NTT provides a computationally efficient isomorphism between these two rings:

$$\widehat{f} := \text{NTT}(f) \in T_q, \quad \text{such that} \quad f \times_{R_q} g = \text{NTT}^{-1}(\widehat{f} \times_{T_q} \widehat{g}),$$

where $\times_{R_q}$ denotes multiplication in R_q and $\times_{T_q}$ the one in T_q, which is a point-wise product. Since multiplication in T_q consists of 128 degree-1 polynomial multiplications, it is substantially more efficient than in R_q, justifying the NTT as an essential rather than optional component of the Kyber algorithm.

NTT Layers and Butterfly Structure

Kyber implements a 7-layer forward NTT using the Cooley-Tukey (CT) algorithm, which recursively breaks down the transform into smaller stages of size 2, known as *butterfly operations*. Each butterfly computes the transformation of two coefficients using a so-called *twiddle factor* (root of unity) to maintain correctness modulo q.

The inverse transform (INTT) is computed using the Gentleman-Sande (GS) algorithm, which is suitable for in-place computation and preserves numerical stability while reversing the NTT. The INTT includes normalization by a factor of n^{-1} mod q and uses twiddle factors in reverse order.

Twiddle Factors

Twiddle factors (also known as *zetas*) are the powers of the primitive root ζ. These values, $\zeta^{2\cdot\text{BitRev}_7(i)+1} \mod q$ used in the degree-1 polynomial multiplications and $\zeta^{\text{BitRev}_7(i)} \mod q$ used in NTT and INTT, are precomputed and stored in a lookup array for efficient access throughout algorithm. The use of precomputed zetas avoids redundant modular exponentiations and improves performance. In optimized versions of Kyber, these twiddle factors can also be stored in alternative modular representations such as Montgomery or Plantard form, which further accelerate modular multiplications during butterfly operations.

Overall, the NTT and INTT procedures in Kyber provide a structured and highly optimized framework for polynomial multiplication, taking advantage of the ring isomorphism $R_q \cong T_q$, precomputed zetas and layered butterfly operations to enable fast and secure cryptographic computations.

3.3.2. Importance of Modular Reduction in NTT Operations

Efficient modular reduction plays a critical role in the performance and correctness of the Number Theoretic Transform (NTT), particularly during the butterfly operations performed in each layer of the algorithm. These steps involve additions, subtractions and multiplications of coefficients modulo the prime $q = 3329$, and the intermediate results of such operations can easily exceed the bounds of the modular ring $\mathbb{Z}_q$.

Without proper modular reduction, these values would no longer be valid elements of the ring R_q, potentially leading to incorrect computations and security vulnerabilities. Therefore, every operation in the NTT and its inverse (INTT) must be followed by a modular reduction to bring the values back within the desired range.

The choice of modular reduction method has a direct impact on performance. Traditional methods like standard integer division are too slow for cryptographic applications, so efficient algorithms like **Montgomery** and **Plantard** reduction are employed. These techniques avoid expensive divisions by replacing them with multiplications and shifts, greatly improving runtime efficiency.

Montgomery reduction, which is the one used in the reference Kyber implementation, transforms operands into a special domain (Montgomery domain) where reductions become more efficient. After performing all the multiplications, the operands are transformed back to the initial domain. On the other hand, Plantard arithmetic, which will be introduced in Section 3.5, presents an alternative approach that can further optimize modular multiplication under certain conditions.

The choice between these reduction strategies is not just an implementation detail but a design decision that affects the entire structure and speed of the NTT. As such, understanding and selecting the appropriate reduction technique is crucial to avoid performance bottlenecks and to enable high-speed polynomial multiplication required in lattice-based cryptographic schemes like Kyber.

3.4. RISC-V Architecture and Relevant Extensions (IMBV)

RISC-V is an open-standard Instruction Set Architecture (ISA) that has gained significant traction in the computing industry due to its flexibility, scalability and cost-effectiveness. Unlike proprietary architectures such as ARM, RISC-V is open-source, allowing organizations to design custom processors without incurring licensing fees (Blog, 2025). This openness fosters innovation and enables tailored solutions across a wide range of applications, from embedded systems to high-performance computing. In contrast, ARM offers pre-designed cores with licensing costs, which, while providing optimized solutions for specific applications, may limit customization options (Stromasys, 2025).

A key feature of RISC-V is its modular design, which allows for a base set of instructions to be extended with standardized or custom extensions, enabling developers to optimize processors for specific workloads. Among these extensions, we highlight the ones we will study in the optimizations of Kyber:

- **Integer (I) Extension**: Provides basic integer computational capabilities, forming the core of the RISC-V ISA.
- **Multiplication and Division (M) Extension**: Adds hardware support for integer multiplication and division operations, improving performance for arithmetic-intensive applications.
- **Bit Manipulation (B) Extension**: Introduces instructions for efficient bit-level operations, such as bitwise rotations and counting leading zeros, which are beneficial in cryptographic algorithms and data processing tasks.
- **Vector (V) Extension**: Enables operations on large datasets by processing multiple data elements in parallel, significantly boosting performance. (Foundation, 2025)

The combination of these extensions allows RISC-V processors to achieve high efficiency in arithmetic computations, making them well-suited for applications requiring intensive mathematical operations, such as cryptographic algorithms. The open and extensible nature of RISC-V, coupled with these powerful extensions, positions it as a compelling choice for developers seeking customizable and high-performance computing solutions.

In the following subsections, we delve deeper into the **Bit Manipulation (B)** and **Vector (V)** extensions, as these play a particularly important role in optimizing

the Kyber implementation. The B extension provides fine-grained control over data at the bit level, which is especially useful in cryptographic routines such as Keccak. The V extension (RVV), on the other hand, enables the use of vectorized instructions for parallel data processing, which can dramatically improve the throughput of operations like the Number Theoretic Transform (NTT). Understanding the capabilities and limitations of these extensions is key to fully exploiting the performance potential of RISC-V platforms in post-quantum cryptographic algorithms.

3.4.1. The Bit Manipulation (B) Extension

The Bit Manipulation (B) extension in RISC-V (EmbedDev, 2025a) introduces a rich set of instructions that improve performance, reduce code size and enable more efficient implementations of low-level operations, many of which are particularly valuable in cryptographic applications. This extension is subdivided into several components, including `Zba`, `Zbb`, `Zbc` and `Zbs`, each focusing on a specific class of operations.

In cryptographic schemes like Kyber, which require numerous bit-level operations for tasks such as Keccak permutation and modular arithmetic, the B extension can provide significant improvements in performance and code compactness.

- **Bitwise rotations:** Instructions like `rol`, `ror` and `rori` (rotate left/right by register or immediate) are essential for efficiently implementing the Keccak-f1600 permutation used in SHA-3 and SHAKE. Without native support, rotations would require multiple shift and logical operations.
- **Bitwise AND-NOT and OR-NOT:** The `andn` and `orn` instructions allow efficient implementation of masking and filtering logic. These are widely used in the χ step of Keccak and in general logic simplifications for finite field arithmetic.
- **Byte-reversal and byte-level operations:** The `rev8` instruction reverses the byte order in a register, which can accelerate serialization, deserialization or endian-conversion tasks common in cryptographic protocol implementations.

These instructions are not only beneficial for increasing performance, but also reduce register pressure and instruction count, crucial aspects when optimizing cryptographic primitives for resource-constrained devices. The B extension is thus a key enabler for high-efficiency cryptographic software on RISC-V platforms.

3.4.2. The Vector (V) Extension (RVV)

The RISC-V Vector Extension (RVV) (EmbedDev, 2025b) is a powerful and flexible instruction set extension designed to accelerate data-parallel operations in RISC-V processors. Unlike traditional SIMD architectures, where the width of vector registers is fixed (e.g., ARM NEON or Intel AVX), RVV introduces vector-length

agnosticism (VLA). This means that code written using RVV can run on any compliant processor regardless of the vector register width (VLEN), making it highly portable and future-proof.

RVV adds 32 vector registers (`v0` through `v31`), each `VLEN`-bit wide. The actual value of `VLEN` is an implementation-specific parameter (commonly 128, 256 or 512 bits), allowing hardware designers to choose an appropriate size for their target performance and area constraints. There are some key parameters in RVV:

- **VLEN** is the hardware-defined width of the vector registers in bits.
- **SEW** (Selected Element Width) defines the bit-width of individual elements operated on within a vector register, such as 8, 16, 32 or 64 bits.
- **LMUL** (Vector Length Multiplier) indicates how many registers are grouped together for a single operation. For instance, `LMUL=2` means two vector registers are used as a single operand group.
- **Tail Policies (ta/tu):** Control what happens to vector elements beyond the active length (`vl`). `ta` (tail agnostic) allows those elements to be overwritten arbitrarily, while `tu` (tail undisturbed) preserves their original values.
- **Mask Policies (ma/mu):** Similarly, `ma` (mask agnostic) allows disabled elements to be overwritten, while `mu` (mask undisturbed) keeps their previous value.

These parameters are configured using the `vsetvli` instruction, which sets the vector length and behavior for subsequent operations. An example of how this instruction is used is as follows:

```
vsetvli x1, x0, e64, m1, ta, mu
```

This sets the `SEW` to 64 bits, `LMUL` to 1 and selects tail-agnostic and mask-undisturbed modes. The register `x1` receives the computed vector length `vl` (number of elements processed) and `x0` holds the total number of elements to process.

3.4.2.1. Instructions and Masking

The RVV instruction set includes a rich variety of operations that are fundamental for implementing data-parallel algorithms, especially in performance-sensitive domains such as cryptography, linear algebra and signal processing. These operations fall into several categories:

- **Arithmetic:** Includes element-wise addition, subtraction, multiplication and division. For example:

```
vadd.vv v1, v2, v3      // v1[i] = v2[i] + v3[i]
vsub.vx v4, v5, x6      // v4[i] = v5[i] - x6
```

- **Logical:** Bitwise operations on elements, including AND, OR, XOR:

```
vand.vv v7, v8, v9      // v7[i] = v8[i] & v9[i]
vxor.vi v10, v11, 0xFF  // v10[i] = v11[i] ^ 0xFF
```

- **Comparison:** Element-wise comparison generates a mask result (stored in a mask register):

```
vmseq.vv v0, v12, v13   // v0[i] = (v12[i] == v13[i])
vmsgtu.vx v0, v14, x7   // v0[i] = (v14[i] > x7)
```

  These comparison instructions are essential when conditionally selecting elements, such as during filtering or branching.

- **Permutations and Data Movement:** Instructions like `vrgather.vv` and `vslideup.vi` allow element shuffling and translation:

```
vrgather.vv v1, v2, v3  // Gather elements from v2
                        // using indices from v3
vslideup.vi v4, v5, 2   // Shift elements in v5 up by
                        // 2, insert 0s at bottom
```

- **Reductions:** RVV supports reduction operations such as sum, max, min:

```
vredsum.vs v6, v7, v8   // v6[0] = v8[0] + v7[1]
                        // + v7[2] + v7[3] + ...
```

  These are used to collapse a vector into a scalar result, useful in norm computations or polynomial coefficient reductions.

As seen in all the examples above, RVV operations are typically available in three forms:

- **.vv**: vector-vector operations where each element is taken from a pair of vector registers.
- **.vx**: vector-scalar operations where one operand is a vector and the other is a scalar from the integer register file.
- **.vi**: vector-immediate operations where the scalar operand is an immediate constant.

These variants provide flexibility and help reduce register pressure, especially when one operand is constant or uniform across elements.

Most vector instructions support optional masking via the `v0` register. When a mask is provided, only those elements for which the corresponding mask bit is set to 1 are modified by the operation. Elements with a mask bit of 0 retain their previous value (mask undisturbed) or may be overwritten arbitrarily (mask agnostic), depending on the policy set in `vsetvli`. For example, consider the following piece of code:

```
vmseq.vi v0, v2, 3          // Create a mask: v0[i] = (v2[i] == 3)
vadd.vv v3, v3, v4, v0.t    // Add v3[i] += v4[i] only where
                            // v2[i] == 3
```

This sequence conditionally adds values from `v4` to `v3`, only where the elements in `v2` are equal to three.

Only `v0` can be used as a mask input in RVV 1.0 for standard instructions. Using `v0` as both a mask and a data register is technically possible but discouraged, as it increases register pressure, especially when `LMUL > 1`.

3.4.2.2. Applications in Cryptography and Kyber

In cryptographic schemes like Kyber, vectorization can significantly accelerate performance, particularly during polynomial arithmetic and modular operations. RVV allows parallel processing of coefficients in polynomials, enabling faster Number-Theoretic Transforms (NTTs), point-wise multiplications and reductions. Operations like `vadd.vv`, `vmul.vv` and `vredsum.vs` align well with common cryptographic workflows. In addition, masking capabilities are critical for implementing constant-time operations to mitigate side-channel attacks.

Overall, RVV provides a highly flexible and scalable vector instruction set architecture, allowing efficient parallel processing of large datasets. Its vector-length agnostic design, support for fine-grained masking and rich set of arithmetic and logical operations make it particularly well suited for implementing high-performance cryptographic algorithms, such as Kyber, on customizable and open RISC-V hardware platforms.

3.5. Modular Multiplication Optimizations: Plantard, HZZ and AMOT

In lattice-based cryptography, modular multiplication plays a critical role in polynomial and matrix operations. A naive approach to modular multiplication would involve directly computing the product of two numbers and then reducing it modulo q. However, this method is computationally expensive, especially when implemented on hardware with limited resources, as it requires costly division operations. Instead, efficient modular reduction techniques are employed to avoid explicit division while

ensuring correctness. The standard approach in Kyber relies on Montgomery modular multiplication, which efficiently reduces large intermediate values without costly division operations. In this work, we explore alternative modular multiplication techniques based on the Plantard method and its extensions to signed arithmetic: HZZ and AMOT. These techniques aim to improve performance while maintaining correctness in modular reduction.

3.5.1. Plantard Modular Multiplication

Plantard's modular multiplication is designed for efficient computation with unsigned integers. The algorithm aims to reduce the computational overhead by minimizing the number of required multiplications and additions. The core idea involves scaling the product of two operands by a precomputed constant $R = P^{-1}\ mod\ 2^{32}$, followed by a series of shifts and multiplications to achieve the modular reduction. Specifically, the algorithm is:

Algorithm 1 Unsigned Plantard Modular Multiplication (Plantard, 2021)

Input: A, B, P, R, n with $0 \leq A, B \leq P$ and $R = P^{-1} \mod 2^{2n}$
Output: C with $0 \leq C < P$ and $C = AB(-2^{-2n}) \mod P$

begin
  $C \leftarrow [([[ABR]_{2n}]^n + 1)\, P]^n$
  if $C = P$ **then**
    return 0
  end if
  return C
end

In the case of Kyber, the modulus $P = q = 3329$, the number of bits required is $n = 16$ and $R = P^{-1}\ mod\ 2^{2n} = 3329^{-1}\ mod\ 2^{32} = 1806234369$. My implementation in C code is:

```
// 0 <= a,b <= Q
// Output c = a * b * (-2^(-32)) mod Q, 0 <= c < Q
static uint16_t unsigned_plantard_mul(uint16_t a, uint16_t b) {

    uint16_t c;

    c = ((uint32_t) a * b * R_UN_PLANT) >> 16; // c = floor( (abR
       mod 2^32) / 2^16 )
    c = ((c + 1) * Q) >> 16; // c = floor( ((c + 1)Q / 2^16 )

    if (c == Q)
        return (uint16_t) 0;
    else
        return c;
}
```

Listing 3.1: Unsigned Plantard Multiplication

This method avoids costly division by directly utilizing bit shifts, improving computational efficiency on hardware architectures such as RISC-V. However, Plantard multiplication is inherently unsigned, which makes it incompatible with Kyber's reference implementation, where signed modular arithmetic is used. This limitation led to the development of signed variants like HZZ and AMOT, which adapt the method for compatibility with Kyber while maintaining efficiency.

3.5.2. Plantard's HZZ Modular Multiplication

The HZZ technique extends Plantard's method to handle signed integers, addressing the limitation of the original algorithm which only supports unsigned inputs. This extension allows for a broader range of applications, particularly in cryptographic algorithms that operate with signed data. The algorithm is as follows:

Algorithm 2 Signed Plantard HZZ Modular Multiplication (Huang et al., 2024)

Input: Two signed integers $A, B \in [-P2^{\alpha}, P2^{\alpha}]$, $P < 2^{n-\alpha-1}$, $R = P^{-1} \ mod^{\pm} \ 2^{2n}$
Output: $r = AB(-2^{-2n}) \ mod^{\pm} \ P$ where $r \in \left[-\frac{P+1}{2}, \frac{P-1}{2}\right]$
$r \leftarrow [([[ABR]_{2n}]^n + 2^{\alpha})\, P]^n$
return r

In the case of Kyber, the modulus $P = q = 3329$, the number of bits required is $n = 16$, we can take $\alpha = 1$ since all numbers modulo q verify the restrictions on a and b and $R = P^{-1} \ mod^{\pm} \ 2^{2n} = 3329^{-1} \ mod^{\pm} \ 2^{32} = 1806234369$. My implementation in C code is:

```
// a,b in [-6658, 6658]
// Output r = a*b*(-2^(-32)) mod+- Q (r in [-(Q + 1)/2, (Q - 1)/2])
static int16_t signed_plantard_mul(int16_t a, int16_t b) {

    int16_t c;

    c = ((int32_t) a * b * R_HZZ) >> 16; // c = floor( (abR mod+-
       2^32) / 2^16 )
    c = ((c + 2) * Q) >> 16;    // c = floor( ((c + 2)Q / 2^16 )

    return c;
}
```

Listing 3.2: Signed Plantard HZZ Modular Multiplication

3.5.3. Plantard's AMOT Modular Multiplication

Similar to HZZ, the AMOT technique also focuses on signed integers. Instead of having floor operations, it makes use of rounding ones. The algorithm is as follows:

Algorithm 3 Signed Plantard AMOT Modular Multiplication (Aoki et al., 2022)

Input: A, B, P, R, n with $|A|, |B| \leq 2^{n-1}$, $R = P^{-1} \ mod^{\pm} \ 2^{2n}$ and $P < 2^{n-1}$
Output: $C = AB(-2^{-2n}) \ mod^{\pm} \ P$ where $r \in \left[-\frac{P-1}{2}, \frac{P-1}{2}\right]$
3: $C \leftarrow \lfloor (\lfloor (ABR \ mod^{\pm} \ 2^{2n}) / 2^{n} \rceil) P / 2^{n} \rceil$
return C

In the case of Kyber, the modulus $P = q = 3329$, the number of bits required is $n = 16$ and $R = P^{-1} \ mod^{\pm} \ 2^{2n} = 3329^{-1} \ mod^{\pm} \ 2^{32} = 1806234369$. My implementation in C code is:

```
// abs(a), abs(b) <= 2^15, Q < 2^15
// Output r = a*b*(-2^(-32)) mod+- Q (r in [-(Q - 1)/2, (Q - 1)/2])
static int16_t signed_plantard_mul(int16_t a, int16_t b) {

    int16_t c;

    c = (((int32_t) a * b * R_AMOT) + (1 << 15)) >> 16; // c =
       round( (abR mod+- 2^32) / 2^16 )
    c = (c * Q + (1 << 15)) >> 16;  // c = round( cQ / 2^16 )

    return c;
}
```

Listing 3.3: Signed Plantard AMOT Modular Multiplication

In a later chapter, I will present a comparison between the results obtained using the reference implementation of Kyber, which relies on Montgomery modular multiplication, and those using the HZZ and AMOT techniques based on Plantard multiplication. The goal is to evaluate whether these alternative methods provide improvements in execution time, cycle counts or overall efficiency. By analysing the performance differences, this comparison will offer insights into the potential benefits of adopting signed Plantard-based modular multiplication for cryptographic computations in resource-constrained environments.

Chapter 4

Development and optimization proposal

4.1. Boards Used in the Experiments: Technical Specifications

This section provides a detailed overview of the development boards used in the experiments conducted in this work. Each board is based on a different RISC-V architecture and serves as a testbed for evaluating cryptographic algorithms and performance optimizations in hardware. The chapter outlines the technical specifications of the RISC-V K230 and Banana Pi F3 boards, highlighting their processor architecture, memory configuration and key hardware features. This information sets the foundation for understanding the capabilities and limitations of each platform, providing context for the performance evaluations presented in subsequent sections.

4.1.1. K230 Board

The first platform used for the experiments in this work is the RISC-V K230 development board, a low-power, Linux-capable platform designed primarily for embedded systems and research in RISC-V computing. This board features a 64-bit processor with a single core and in-order execution, making it particularly suitable for benchmarking cryptographic algorithms in constrained environments. This design simplifies analysis but may restrict peak throughput compared to out-of-order or superscalar designs.

The processor used in the K230 is the **Xuantie C908**, operating at 1.6 GHz. It supports both RV64 and RV32 execution modes and features a 9-stage pipeline with in-order fetch, dispatch, execute and retire. For scalar instructions, the core supports dual-issue execution, while vector instructions are issued singly. It also supports concurrent bus access for memory read/write operations with up to 8-way reads and 12-way writes and includes features such as write combining for enhanced memory performance. The cache hierarchy includes 32 KB of L1 instruction cache, 32 KB of L1 data cache and a 256 KB L2 cache (Liu, 2022).

The processor implements the RISC-V `rv64imafdcvxthead` instruction set ar-

chitecture (ISA), which includes:

- **I**: Base integer instructions
- **M**: Integer multiplication and division
- **A**: Atomic operations
- **F/D**: Single- and double-precision floating-point arithmetic
- **C**: Compressed instructions
- **V**: RISC-V Vector Extension version 1.0, with a vector register length (VLEN) of 128 bits
- **xthead**: Custom T-Head extensions for performance optimization and workload-specific enhancements

Memory management is handled by an MMU implementing the `Sv39` virtual memory scheme, which allows a 39-bit virtual address space. The board includes approximately 479 MiB of usable RAM (491,384 kB detected) and a 4.0 GiB swap partition.

On the software side, the K230 runs a Debian-based Linux distribution and supports development with a RISC-V-compatible GCC toolchain (version 13.2.0). A wide array of RISC-V development utilities and libraries are available through the system package manager.

Remote access is configured via a secure workflow. A VPN connection is first established with the university's campus network, after which SSH access is made to an intermediate server (`sardina.dacya.ucm.es`). From there, the development board is accessed via its local network address (`k230.lan`), providing a stable and secure environment for development and testing. File transfers were performed using the same route, leveraging `sftp` over the SSH connection to ensure secure transmission of data to and from the board.

4.1.2. Banana Pi F3 Board

The second platform used for the experiments is the Banana Pi F3, a modern RISC-V development board designed to balance performance and efficiency for edge computing and embedded workloads. It is equipped with a **SpacemiT X60** processor, an octa-core 64-bit RISC-V chip targeting high-performance applications such as SBCs, storage servers, robotics and industrial control systems.

Each of the eight cores operates at frequencies between 614 MHz and 1.6 GHz, with performance scaling up to 2.0 GHz in higher-performance variants. The microarchitecture follows an 8-stage, in-order dual-issue pipeline design. It supports dual-issue for both scalar and vector read/write instructions and employs a dual-cache structure (Cache + TCM) along with intra- and inter-cluster coherence, indicating a multi-cluster, multi-core architecture (Corporation, 2024).

The X60 adheres to the RVA22 profile and implements the RISC-V Vector Extension version 1.0 with a 256-bit vector register length (VLEN). It provides broad support for integer (INT8/16/32/64) and floating-point (BF16/FP16/FP32/FP64) data types. Additionally, it offers 2 TOPS@INT8 AI performance via compliance with RISC-V IME extensions, making it especially suitable for workloads involving both classical and AI-enhanced computation.

The processor supports the following RISC-V instruction set extensions:

- **I**: Base integer instructions
- **M**: Integer multiplication and division
- **A**: Atomic operations
- **F/D**: Single- and double-precision floating-point arithmetic
- **C**: Compressed instructions
- **V**: Vector Extension version 1.0 with 256-bit VLEN
- **Zba, Zbb, Zbs**: Bit manipulation extensions
- **Zvfh, Zvkt**: Vector floating-point and cryptographic extensions
- **Zkt**: Scalar cryptographic instructions

The system includes 8016 MiB (approximately 7.6 GiB) of usable RAM and does not use swap. This generous memory allocation supports multi-threaded cryptographic benchmarks and parallel execution of test suites.

The board runs **Bianbu 2.1**, a Debian-based Linux distribution tailored for RISC-V and includes GCC 13.2.0 as the primary compiler, along with a suite of RISC-V libraries and tools.

Remote access is managed similarly to the K230: after connecting to the campus network via VPN, the board is accessed via SSH through an intermediate server, using the local network address `artecs-BPF3.lan`. This ensures a consistent testing setup across all experimental platforms. File transfers were also carried out through the same path using `sftp`, ensuring secure and reliable data exchange with the board.

4.2. Initial Study and Analysis of Existing Implementations

This section provides a comparative analysis between the Kyber reference implementation and an optimized version specifically tailored for RISC-V platforms. The goal is to identify improvements made in the optimized version and understand the rationale behind those enhancements, focusing on RV64-based systems, which are used in the experimental platforms of this study.

The **reference implementation** of Kyber, obtained from the NIST PQC project website[1], is a clean and portable C implementation designed for correctness and standardization compliance. It does not leverage architecture-specific optimizations and uses conventional Montgomery modular multiplication for its Number Theoretic Transform (NTT) operations.

In contrast, the **optimized implementation**, obtained from the open-source repository accompanying the work in (Zhang et al., 2024), targets RV{32,64}IM{B}{V} instruction sets and is specifically tuned for the XuanTie C908 core and for VLEN = 128. This version integrates several enhancements in both modular arithmetic and overall computational structure. In the next subsections, an in depth explanation of all the optimizations that were made on Kyber for RV64 platforms is given.

4.2.1. Keccak Optimizations

4.2.1.1. Keccak on RV64I

Firstly, Keccak-f1600 requires several 64-bit left rotation operations, particularly in the θ and $\rho\pi$ steps. Since the base RV64I instruction set lacks a dedicated rotate instruction, such operations are emulated using a sequence of shift and XOR instructions: first, a left shift followed by a right shift of the complementary amount and finally an XOR to combine the two shifted values. Specifically, the code is as follows:

```
slli t, a, n
srli b, a, 64 - n
xor  b, b, t
```

This emulation typically requires two temporary registers per rotation: one for intermediate computation and another for storing the result.

Secondly, to address limitations of the RV64I instruction set in the χ step, a **lane complementing** technique is used. Instead of computing bitwise operations directly, a subset of the 64-bit lanes is negated during load and store stages. This optimization reduces the number of explicit NOT operations from 25 to 8 and transforms several AND operations into OR operations, introducing new and more efficient instruction patterns.

Thirdly, the state of Keccak-f1600 is held in 25 registers. With only 30 general-purpose registers available, the remaining five must accommodate constants, address pointers and intermediate values. To avoid unnecessary memory operations, an **in-place register allocation** strategy is adopted. This approach shifts a subset of the state registers during the $\rho\pi$ steps and the results are restored to their original positions during χ, avoiding register-to-register moves or spilling to memory.

Finally, optimizing for **dual-issue pipelines** on RV64I requires careful scheduling to mitigate hazards, especially read-after-write (RAW) dependencies. In contrast

[1] Available at `https://csrc.nist.gov/CSRC/media/Projects/post-quantum-cryptography/documents/round-3/submissions/Kyber-Round3.zip`

to prior work on ARMv8-A, where barrel shifters and additional registers eased these issues, the limited register file on RV64I increases the likelihood of spills and stalls. A detailed allocation strategy is employed for the θ step, the most register-intensive part of a round. By carefully sequencing the computation of intermediate variables and releasing registers when no longer needed, most RAW hazards are avoided. Nevertheless, a single additional temporary register is required, obtained through **stack spilling**. This trade-off avoids pipeline stalls but introduces two extra memory instructions per round.

Performance comparisons reveal that using stack spilling to avoid RAW hazards results in improved execution metrics. Specifically, the optimized implementation reduced both cycle count and CPI while slightly increasing instruction count due to load/store operations.

4.2.1.2. Keccak on RV64IB

The RV64IB extension introduces two key instructions: `rori` for bitwise rotations and `andn` for computing the bitwise AND between one operand and the negation of another. These instructions directly implement the operations needed for the θ and χ steps, reducing the instruction count and eliminating the need for temporary registers used in RV64I. As a result, techniques such as lane complementing become unnecessary. The use of these two instructions also eliminates the need for stack spilling. This allows the implementation to maintain dual-issue throughput while avoiding pipeline stalls completely.

4.2.1.3. Keccak on RVV

The vectorized implementation closely mirrors the scalar version. All logical operations in RV64I have corresponding vector counterparts. Given a vector length (VLEN) of 128 bits, the vector backend can process two Keccak-f1600 instances in parallel.

RVV provides 32 vector registers, which simplifies scheduling and eliminates the need for spilling. However, some shift operations in Keccak-f1600 require larger immediates than the 5-bit field supported by `vsll.vi`. This requires the use of `vx` variants, which take a scalar register as input. Consequently, scalar instructions like `li` are occasionally required.

4.2.1.4. Hybrid Implementations

Hybrid approaches combine scalar and vector execution to leverage parallelism in processors with independent scalar and SIMD pipelines. This technique has been used in other cryptographic algorithms. However, on the current platform its effectiveness varies based on the architecture.

For RV64I with vector and bit-manipulation extensions, the hybrid method offers little to no improvement. The scalar-only implementation already achieves near-optimal CPI by fully utilizing the execution pipeline. Although vector instructions

are introduced to exploit parallelism, they do not provide a performance improvement in this case. This is because the vector backend of the processor is relatively weak and cannot execute the bitwise operations that dominate Keccak much faster than the scalar pipeline. Moreover, issuing vector instructions adds pressure to the front-end, which must now handle both scalar and vector instruction streams. As a result, increased front-end contention reduces overall throughput without compensating benefits from vector execution. For these reasons, the hybrid design is better suited for architectures with limited scalar performance and stronger vector back-ends.

4.2.2. Modular Arithmetic Optimizations

The reference Kyber implementation uses signed Montgomery multiplication for modular reduction, which is well-suited for general-purpose processors. In the optimized version, while Montgomery arithmetic remains the default choice for RV64 platforms, Plantard arithmetic is also explored due to its ability to reduce the number of multiplications when multiplying by twiddle factors in the NTT. However, due to the lack of hardware support for the rounding operation required in one of the newer Plantard algorithms (Algorithm 4 in Zhang et al. (2024)), Montgomery arithmetic is still preferred for RV64 platforms.

Signed Montgomery multiplication, as described in (Seiler, 2018), minimizes register usage by requiring only single-width operations and fewer high-latency instructions, which makes it compatible with the dual-issue architecture of the C908 core. This contrasts with the Plantard arithmetic variant that demands wide multiplications and explicit rounding, which are more costly in terms of execution cycles on RV64.

4.2.3. NTT Optimizations

4.2.3.1. NTT on RV64IM

On 64-bit scalar cores with dual-issue in-order pipelines, such as the Xuantie C908, one of the primary bottlenecks in NTT arises from Read-After-Write (RAW) hazards between sequential arithmetic operations. To mitigate this, the implementation alternates multiple butterfly operations to increase instruction-level parallelism. Although theoretical throughput would allow up to 6–8 concurrent CT (Cooley and Tukey, 1965) butterflies, register pressure on RISC-V limits practical implementations to four-way interleaving.

To improve performance further, the `mulw` instruction is employed in place of `mul`, as `mulw` is optimized for 64-bit cores when dealing with 32-bit modular arithmetic. The layer interleaving strategy adopted is 3+3+1 rather than the more aggressive 4+3 used in RV32IM, due to the increased register demand introduced by Montgomery multiplication constants. This design maintains a balance between pipeline utilization and register availability.

4.2.3.2. NTT on RVV

Vectorized implementations of Kyber's NTT benefit from the RISC-V Vector Extension (RVV). Unlike scalar cores, RVV lacks native 16×32-bit multiplication instructions, making Plantard arithmetic inefficient due to reduced parallelism. As a result, Montgomery multiplication is preferred for vector NTTs.

The core optimization involves using the `vmul{h}.vx` instruction, which multiplies a vector by a scalar. By storing constants in scalar registers and using scalar loads (which are cheaper than vector loads), register pressure is reduced and throughput is improved. The chosen layer merging strategy for NTT is 1+6, which minimizes intermediate modular reductions.

During inverse NTT (INTT), normalization steps include multiplying by a Montgomery constant ($\frac{\text{MONT}^2}{128}$) and performing additional modular reductions after specific layers to prevent overflow in vector registers. In total, 18 extra modular reductions are applied during INTT. Instructions like `vrgather.vv` and `vmerge.vvm` assist in data rearrangement within vector registers to maintain logical polynomial order during computation.

4.2.3.3. LMUL Settings on RVV

The vector length multiplier (LMUL) controls how many elements are processed in parallel. Analysis indicates that increasing LMUL (e.g., LMUL > 1) does not improve performance in core NTT and INTT routines, due to instruction dependencies and data reuse patterns. However, related subroutines such as `poly_reduce` and `poly_tomont` show modest gains when LMUL is increased.

4.2.3.4. Hybrid Implementations

Combining scalar and vector instructions in hybrid NTT designs can, in theory, offer the best of both worlds. However, experiments done in the work (Zhang et al., 2024) on the C908 core suggest that this approach does not yield significant performance benefits. Execution patterns such as `vmul; vmul; vmul; mul` and `vmul; vmul; vmul` have the same latency, indicating that a 4:1 ratio of vector-to-scalar instructions is optimal for C908. Implementing hybrid NTTs would require constructing multi-way interleaved loops, like for example 33-, 17- or 9-way NTTs for 16-, 32- and 64-bit operands, respectively. These techniques may prove more effective on other platforms, such as those used for Dilithium or schemes like Raccoon, which involve 64-bit NTTs and benefit from different vector-to-scalar balance.

These improvements are particularly valuable for performance evaluation on RISC-V 64-bit platforms. Further comparisons and quantitative results are presented in Chapter 5.

4.3. Integration of Plantard Modular Multiplication

To evaluate the performance of alternative modular multiplication techniques, the reference implementation of Kyber was modified to integrate the Plantard method. This required systematic changes to the original codebase, particularly in the modular reduction, polynomial arithmetic and Number Theoretic Transform (NTT) routines.

The first modification involved removing the Montgomery reduction function: `int16_t montgomery_reduce(int32_t a)`, which was used in the original implementation to reduce intermediate multiplication results. In its place, a new function `int16_t signed_plantard_mul(int16_t a, int16_t b)` was introduced to implement signed Plantard modular multiplication. Additionally, the helper function `int16_t to_plant(int16_t a)` was added to convert values to the Plantard domain. Both functions were defined in the `reduce.c` and `reduce.h` files, along with relevant constants:

```
#define R_AMOT 1806234369 // 3329^{-1} mod 2^{32}
#define ENTER_PLANT -341  // 2^{64} mod 3329
```

To reflect these changes in the polynomial layer, the function `void poly_tomont (poly *r)` was removed and replaced with `void poly_toplant(poly *r)`, which transforms polynomial coefficients into the Plantard domain. All calls to the former function throughout the implementation were replaced accordingly.

In the NTT module (`ntt.c`), the modular multiplication function `fqmul` was updated to utilize `signed_plantard_mul`. The constant used in the inverse NTT function `intt` was also replaced with:

```
const int16_t f = 2208; // ENTER_PLANT / 128 mod q
```

The last and most substantial modification involved recomputing the array of zeta constants used in the NTT and inverse NTT. These values, which represent roots of unity in the finite field, must be precomputed in Plantard representation to ensure correctness and consistency across all modular operations. To generate these constants, a dedicated C program named `gen_zetas.c` was written. This program computes the zetas in the Plantard domain, applies the appropriate bit-reversal permutation and outputs the values in a format ready for inclusion in the Kyber codebase. The program and its output are included in Appendix A.

After making any modifications, tests from the reference implementation were conducted to ensure that the algorithm continued to function correctly. These tests verified that the output vectors produced by all the main Kyber functions matched the expected results.

These modifications enable the Kyber implementation to perform all modular arithmetic within the Plantard domain, facilitating a fair comparison with the original Montgomery-based implementation in terms of performance and efficiency across multiple platforms.

Chapter 5

Results and discussion

5.1. Performance Comparison: Reference vs. Optimized Implementation

This section presents a performance comparison between the Kyber reference implementation and the optimized version on the K230 and the Banana Pi F3 boards. The evaluation focuses exclusively on the improvements achieved through software optimization, without inter-platform performance comparisons, which are addressed in a later section.

The optimized implementation incorporates advanced techniques such as improved modular arithmetic, a refined Number Theoretic Transform (NTT) and a more efficient SHA-3 integration, all of which were described in Section 4.2. These enhancements are designed to reduce execution time and cycle counts, thereby significantly improving the efficiency of Kyber's key generation, encryption and decryption routines.

5.1.1. Results Overview

Tables 5.1 and 5.2 summarize the performance results for both implementations on each board. Each result represents the average time and cycle count from 10,000 iterations of each Kyber operation.

Table 5.1: Performance comparison: Reference vs. Optimized on K230.

Kyber Version	**Reference**	**Optimized**
Kyber512	9742 ms / 15,584M	2156 ms / 3451M
Kyber768	15384 ms / 24,613M	3395 ms / 5433M
Kyber1024	22285 ms / 35,653M	5135 ms / 8216M

Table 5.2: Performance comparison: Reference vs. Optimized on Banana Pi F3.

Kyber Version	Reference	Optimized
Kyber512	9103 ms / 14,565M	1849 ms / 2960M
Kyber768	14467 ms / 23,147M	2935 ms / 4694M
Kyber1024	20889 ms / 33,419M	4440 ms / 7105M

5.1.2. Quantitative Speed-up Analysis

To quantify the performance gains achieved by the optimized implementation, Table 5.3 reports the speed-up factors, computed as the ratio of the reference execution time to that of the optimized version.

Table 5.3: Speed-up Factors of Optimized vs. Reference Implementation

Kyber Variant	K230	Banana Pi F3
Kyber512	4.52×	4.92×
Kyber768	4.53×	4.93×
Kyber1024	4.34×	4.71×

5.1.3. Performance Analysis

The data confirms a substantial performance boost with the optimized implementation across all Kyber variants. On both platforms, the optimized code consistently delivers speed-ups exceeding 4x over the reference version. As explained in Section 4.2, these gains are attributable to:

- **Efficient Modular Arithmetic:** Arithmetic operations were tailored to the processor's capabilities, reducing overhead and improving performance across key routines.
- **Optimized NTT Calculation:** Techniques like instruction interleaving and careful register management enhanced throughput in transform steps critical to Kyber.
- **Improved SHA-3 Integration:** The Keccak-f1600 permutation was adapted to the RISC-V architecture using instruction-level optimizations and pipeline-aware scheduling.

These improvements lead to a dramatic reduction in both execution time and cycle counts, making the optimized implementation far more suitable for performance-sensitive environments.

5.1.4. VLEN Optimization Limitations

Despite the architectural differences between the two platforms, including the number of cores and the vector register length, the optimized implementation yields very similar absolute execution times on both the K230 and the Banana Pi F3.

This unexpected parity can be explained by design decisions in the source of the optimized implementation.

As stated in Section 4.2, the authors of the optimized Kyber implementation focused specifically on **RISC-V cores with VLEN = 128**. The design and tuning of their code did not account for vector units with longer register lengths such as VLEN = 256. Although the Banana Pi F3 supports VLEN = 256 and a more advanced multi-core architecture, the implementation is unable to exploit the potential advantages of this wider vector length. Consequently, its performance remains close to that of the single-core, in-order K230 processor, which features VLEN = 128.

This shows the importance of adjusting vector-based optimizations for different RISC-V platforms. If the code is not adapted to make use of larger vector units, much of the hardware's potential goes unused.

5.2. Impact of Plantard Multiplication on Performance

This section evaluates the impact of using Plantard modular multiplication in the Kyber implementation, specifically analysing two variants: HZZ and AMOT. These were integrated into the reference implementation, replacing the standard Montgomery multiplication. The detailed implementation process was described in Section 4.3. This analysis aims to determine whether Plantard arithmetic can offer performance advantages over the traditional Montgomery method, particularly in the context of the RISC-V platforms used in this study.

5.2.1. Results Overview

Tables 5.4 and 5.5 show the execution times and cycle counts for the HZZ and AMOT versions of Plantard multiplication on the K230 and Banana Pi F3 boards. For comparison, the results of the reference implementation using Montgomery multiplication are also included.

Table 5.4: Performance comparison: Plantard (HZZ and AMOT) vs. Reference on K230.

Kyber Version	Reference	HZZ	AMOT
Kyber512	9742 ms / 15,584M	9653 ms / 15,439M	9867 ms / 15,786M
Kyber768	15384 ms / 24,613M	15240 ms / 24,383M	15568 ms / 24,908M
Kyber1024	22285 ms / 35,653M	22144 ms / 35,425M	22578 ms / 36,119M

Table 5.5: Performance comparison: Plantard (HZZ and AMOT) vs. Reference on Banana Pi F3.

Kyber Version	Reference	HZZ	AMOT
Kyber512	9103 ms / 14,565M	8973 ms / 14,359M	9163 ms / 14,661M
Kyber768	14467 ms / 23,147M	14232 ms / 22,762M	14742 ms / 23,585M
Kyber1024	20889 ms / 33,419M	20605 ms / 32,956M	21057 ms / 33,675M

5.2.2. Relative Performance Gains

To further illustrate the performance differences, Table 5.6 provides a percentage comparison of the execution time of the HZZ and AMOT variants relative to the reference implementation.

Table 5.6: Relative performance of Plantard multiplication vs. Reference (percentage of reference execution time).

Kyber Version	K230 HZZ	K230 AMOT	Banana Pi F3 HZZ	Banana Pi F3 AMOT
Kyber512	99.09%	101.28%	98.57%	100.66%
Kyber768	99.06%	101.20%	98.38%	101.90%
Kyber1024	99.37%	101.31%	98.63%	100.80%

5.2.3. Performance Analysis

The results indicate that the HZZ variant of Plantard multiplication offers a slight improvement in performance compared to the reference implementation using Montgomery multiplication. However, this improvement is marginal and the gains are much lower than initially expected. Specifically:

- **HZZ Variant:** It provides a minor speed-up compared to the Montgomery version, typically around 1-2% depending on the Kyber parameter set and the board used. This improvement is consistent across platforms and Kyber versions but modest. These modest improvements can be attributed to the reduced multiplication count in Plantard arithmetic, but the actual benefit is constrained by architectural factors. On the K230 board, which features a single in-order core with limited instruction-level parallelism, the bottlenecks of memory access and scalar throughput likely diminish the impact of fine-grained arithmetic optimizations. On the Banana Pi F3, despite its more capable 8-core design and wider vector unit, the marginal gains suggest that the Plantard approach does not significantly benefit from the available hardware parallelism.
- **AMOT Variant:** It performs slightly worse than the reference implementation. The AMOT variant includes extra rounding operations, which, despite being designed for improved accuracy, introduce enough overhead to negate any potential performance gain.

Overall, these findings suggest that while the HZZ variant can slightly improve performance, the expected theoretical gains from Plantard multiplication are not

fully realized on the current hardware platforms. This highlights the importance of aligning algorithmic optimizations with the specific architectural strengths and limitations of the target processor.

5.3. Performance Comparison Across Boards

This section provides a comparative analysis of the performance of Kyber on the two distinct RISC-V platforms K230 and Banana Pi F3. These boards differ significantly in their architectural characteristics, which directly impact the performance of cryptographic computations. The results presented here aim to highlight how these architectural differences influence execution time and cycle counts for the different Kyber versions.

5.3.1. Results Overview

Tables 5.7 and 5.8 present the execution times and cycle counts for Kyber512, Kyber768 and Kyber1024 across the two platforms. Both the reference and optimized implementations are included for a comprehensive comparison.

Table 5.7: Execution Time Comparison (in milliseconds) Across Platforms.

Implementation	Kyber512	Kyber768	Kyber1024
K230 (Reference)	9742	15384	22285
Banana Pi F3 (Reference)	9103	14467	20889
K230 (Optimized)	2156	3395	5135
Banana Pi F3 (Optimized)	1849	2935	4440

Table 5.8: Cycle Count Comparison (in millions) Across Platforms.

Implementation	Kyber512	Kyber768	Kyber1024
K230 (Reference)	15,584	24,613	35,653
Banana Pi F3 (Reference)	14,565	23,147	33,419
K230 (Optimized)	3,452	5,433	8,216
Banana Pi F3 (Optimized)	2,961	4,694	7,105

5.3.2. Performance Analysis

The performance results reveal clear advantages for the Banana Pi F3 over the K230 across all Kyber parameter sets and implementations. This disparity can be directly linked to the architectural differences between the two platforms.

- **Core and Pipeline Design:** The K230 features a single-core, dual-issue in-order processor (Xuantie C908), while the Banana Pi F3 uses the SpacemiT X60, an 8-core processor with a dual-issue in-order pipeline. Although both are in-order designs, the multi-core nature of the Banana Pi F3 and its more

modern microarchitecture allow it to more efficiently handle cryptographic workloads.

- **Vector Capabilities:** The Banana Pi F3 supports the RISC-V Vector Extension (RVV) v1.0 with a vector length (VLEN) of 256 bits, whereas the K230 supports only 128-bit vectors. This gives the Banana Pi F3 board a theoretical advantage in vector throughput, especially in the context of heavily vectorized cryptographic routines like Kyber. However, the Kyber implementation used in these experiments was specifically optimized for a fixed `VLEN` of 128 bits. As a result, it does not fully exploit the 256-bit capability of the Banana Pi F3, limiting the extent of its performance advantage.
- **Clock Speed and Resource Availability:** While both platforms operate at frequencies up to 1.6 GHz, the Banana Pi F3 can dynamically scale up to 2.0 GHz under load. Moreover, its significantly larger RAM capacity (8016 MiB vs. 479 MiB on the K230) reduces pressure on memory subsystems.

The optimized implementation yields a reduction in execution time of approximately 77–79% on both platforms. While the Banana Pi F3 performs better in absolute terms, for example optimized Kyber1024 executes in 4440 ms on the Banana Pi F3 vs. 5135 ms on the K230, the performance gap is less dramatic than might be expected given the Banana Pi F3's architectural advantages. This is largely due to the optimization being tailored for 128-bit vector widths, which prevents the Banana Pi F3 from leveraging its full 256-bit vector processing potential.

Cycle counts reinforce these observations. The Banana Pi F3 consistently requires fewer cycles than the K230, indicating higher overall efficiency per instruction. Nonetheless, the relative gains could be even more significant with a vector implementation adapted to wider VLENs.

In summary, both platforms benefit substantially from vectorized and dual-issue optimizations, but the Banana Pi F3 has more architectural headroom that is not fully utilized due to the fixed `VLEN=128` design.

Chapter 6

Conclusions and Future Work

6.1. Summary of Contributions and Achievements

This project has focused on the efficient implementation and optimization of the post-quantum cryptographic algorithm Kyber on RISC-V architectures. The work carried out can be summarized in several key contributions:

1. **Comprehensive Study of Kyber and Modular Arithmetic:** The project began with an in-depth study of classical and post-quantum cryptography, focusing specifically on lattice-based schemes. A thorough understanding of the Learning With Errors (LWE) problem and its variants, including Module-LWE, was developed. This theoretical foundation was essential for understanding the security basis of Kyber.

2. **Analysis and Optimization of Kyber:** Two implementations of Kyber were considered for analysis: the reference version published by NIST and an optimized version specifically designed for RISC-V. The optimized version integrates a variety of enhancements, including efficient Keccak (SHAKE) optimizations and improved Number Theoretic Transform (NTT) for polynomial multiplication, taking advantage of RV64I, RV64IB and RVV extensions.

3. **Integration of Plantard Arithmetic:** The project also explored alternative modular multiplication techniques, specifically Plantard arithmetic. Both the HZZ and AMOT variants of Plantard multiplication were implemented and integrated into the reference version of Kyber. This allowed for a direct comparison between Montgomery multiplication, used in the reference, and Plantard multiplication, providing insights into their relative efficiency.

4. **Performance Evaluation on Multiple Platforms:** Extensive benchmarking was performed on two distinct RISC-V platforms: the K230 and Banana Pi F3 boards. These platforms differ significantly in their architectural characteristics, with the K230 featuring a single-core, in-order, single-issue design, while the Banana Pi F3 is an out-of-order, multi-core, dual-issue processor. The performance of both the reference and optimized Kyber implementations

was evaluated in terms of execution time and cycle counts. Additionally, the impact of Plantard multiplication was analysed.

5. **Consistent Measurement Methodology:** To ensure reliable and comparable results, the same measurement methodology was applied across all platforms. Execution time was measured using the `gettimeofday` command, while cycle counts were obtained using the `perf` tool. This consistency in measurement is critical for drawing accurate conclusions about the performance of the different implementations.

6. **Insightful Analysis of Results:** The project provided a detailed analysis of the obtained results, highlighting the architectural factors that influenced performance on each platform. It demonstrated how the optimizations not only improved the performance of Kyber but also scaled effectively across platforms with very different characteristics.

Overall, this project has successfully shown the feasibility of optimizing post-quantum cryptography on RISC-V, providing a clear understanding of the impact of different optimization techniques and hardware architectures on cryptographic performance.

6.2. Future Work

This project has shown the efficient implementation of Kyber on RISC-V platforms, but several areas remain for further exploration:

- **Alternative Modular Arithmetic Techniques:** Beyond Plantard multiplication, other methods such as Barrett or optimized Montgomery multiplication could be tested, potentially further improving performance.
- **Additional RISC-V Extensions:** Exploring other RISC-V extensions, such as scalar cryptography (Zk), could lead to more optimized implementations of Kyber.
- **Energy Efficiency Analysis:** Future work could focus on measuring and optimizing the energy consumption of Kyber on RISC-V, providing insights into its suitability for resource-constrained devices.
- **Leveraging Matrix Instructions:** The RISC-V Matrix Extension specification (v0.5b) (RISC-V International, 2025) introduces matrix registers and operations, which could significantly accelerate the matrix-vector multiplications in Kyber. Although this extension is still under development and lacks hardware support, future implementations could explore its potential to optimize the polynomial arithmetic in Kyber. Until physical implementations become available, the code could be tested on RISC-V simulators supporting this extension.

These directions offer a pathway to further enhance the performance, security and versatility of Kyber on RISC-V platforms.

Bibliography

Aoki, D., Minematsu, K., Okamura, T. and Takagi, T. Efficient word size modular multiplication over signed integers. In *2022 IEEE 29th Symposium on Computer Arithmetic (ARITH)*, 94–101. 2022.

Bernstein, D., Buchmann, J. and Dahmen, E. *Post-Quantum Cryptography*. Springer Berlin Heidelberg, 2009. ISBN 9783540887027.

Blog, P. Risc-v vs. arm: Who wins in 8 categories? *Paessler Blog*, 2025.

Buchmann, J. *Introduction to Cryptography*. Undergraduate Texts in Mathematics. Springer, 2nd edn., 2004.

Cooley, J. W. and Tukey, J. W. An algorithm for the machine calculation of complex fourier series. *Mathematics of Computation*, Vol. 19, 297–301, 1965.

Corporation, S. Spacemit x60 core. `https://www.spacemit.com/en/spacemit-x60-core/`, 2024.

EmbedDev, F. Risc-v bitmanip extension version 1.0.0. `https://five-embeddev.com/riscv-bitmanip/1.0.0/bitmanip.html`, 2025a.

EmbedDev, F. Risc-v v vector extension specification, version 1.0. `https://five-embeddev.com/riscv-v-spec/v1.0/v-spec.html`, 2025b.

Foundation, R.-V. Risc-v "v" vector extension. 2025.

Grover, L. K. A fast quantum mechanical algorithm for database search. 1996.

Huang, J., Zhao, H., Zhang, J., Dai, W., Zhou, L., Cheung, R. C. C., Koç, Ç. K. and Chen, D. Yet another improvement of plantard arithmetic for faster kyber on low-end 32-bit iot devices. *IEEE Transactions on Information Forensics and Security*, Vol. 19, 3800–3813, 2024.

Liu, C. Xuantie c908: High-performance risc-v processor catered to aiot industry. 2022.

Mavroeidis, V., Vishi, K., Zych, M. D. and Jøsang, A. The impact of quantum computing on present cryptography. *CoRR*, Vol. abs/1804.00200, 2018.

National Institute of Standards and Technology. SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions. 2015.

National Institute of Standards and Technology. FIPS 203: Module-Lattice-Based Key-Encapsulation Mechanism (ML-KEM). 2024.

Plantard, T. Efficient word size modular arithmetic. *IEEE Transactions on Emerging Topics in Computing*, Vol. 9, 1506–1518, 2021.

Regev, O. On lattices, learning with errors, random linear codes, and cryptography. *Journal of the ACM*, Vol. 56(6), 34:1–34:40, 2009.

Regev, O. The learning with errors problem. *Invited survey in CCC*, Vol. 7(30), 11, 2010.

RISC-V International. Risc-v matrix extension specification v0.5b (64-bit encoding). `https://lists.riscv.org/g/tech-attached-matrix-extension/attachment/210/1/riscv-matrix-spec-v0.5b-64bit-encoding.pdf`, 2025.

Seiler, G. Faster AVX2 optimized NTT multiplication for ring-LWE lattice cryptography. Cryptology ePrint Archive, Paper 2018/039, 2018.

Shor, P. W. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM Journal on Computing*, Vol. 26(5), 1484–1509, 1997. ISSN 1095-7111.

Stromasys. Risc-v vs arm - a strategic guide to processor architecture. *Stromasys*, 2025.

Wang, Y. and Wang, M. Module-LWE versus ring-LWE, revisited. Cryptology ePrint Archive, Paper 2019/930, 2019.

Zhang, J., Yan, Y., Huang, J. and Çetin Kaya Koç. Optimized software implementation of keccak, kyber, and dilithium on RV32,64IMBV. Cryptology ePrint Archive, Paper 2024/1515, 2024.

Appendix A

Zetas Generation

A.1. Zeta Generation Program: `gen_zetas.c`

```c
#include <stdint.h>
#include <stdio.h>

#define KYBER_Q 3329
#define KYBER_ROOT_OF_UNITY 17
#define PLANT -1353 // -2^32 mod Q
#define R_HZZ 1806234369 // Q^-1 mod 2^32

static const uint8_t tree[128] = {
  0, 64, 32, 96, 16, 80, 48, 112, 8, 72, 40, 104, 24, 88, 56, 120,
  4, 68, 36, 100, 20, 84, 52, 116, 12, 76, 44, 108, 28, 92, 60,
     124,
  2, 66, 34, 98, 18, 82, 50, 114, 10, 74, 42, 106, 26, 90, 58, 122,
  6, 70, 38, 102, 22, 86, 54, 118, 14, 78, 46, 110, 30, 94, 62,
     126,
  1, 65, 33, 97, 17, 81, 49, 113, 9, 73, 41, 105, 25, 89, 57, 121,
  5, 69, 37, 101, 21, 85, 53, 117, 13, 77, 45, 109, 29, 93, 61,
     125,
  3, 67, 35, 99, 19, 83, 51, 115, 11, 75, 43, 107, 27, 91, 59, 123,
  7, 71, 39, 103, 23, 87, 55, 119, 15, 79, 47, 111, 31, 95, 63, 127
};

int16_t zetas[128];

int16_t signed_plantard_mul(int16_t a, int16_t b) {
  int16_t c;
  c = ((int32_t) a * b * R_HZZ) >> 16;
  c = ((c + 2) * KYBER_Q) >> 16;
  return c;
}

void gen_zetas() {
  unsigned int i;
  int16_t tmp[128];

  tmp[0] = PLANT;
```

```
34    for(i=1;i<128;i++)
35      tmp[i] = signed_plantard_mul(tmp[i-1], PLANT *
            KYBER_ROOT_OF_UNITY % KYBER_Q);
36
37    for(i=0;i<128;i++) {
38      zetas[i] = tmp[tree[i]];
39      if(zetas[i] > KYBER_Q/2)
40        zetas[i] -= KYBER_Q;
41      if(zetas[i] < -KYBER_Q/2)
42        zetas[i] += KYBER_Q;
43    }
44  }
45
46  int main() {
47    gen_zetas();
48    printf("const int16_t zetas[128] = {");
49    for(int i = 0; i < 127; ++i) {
50      if(i % 8 == 0)
51        printf("\n\t");
52      printf("%d, ", zetas[i]);
53    }
54    printf("%d\n};\n", zetas[127]);
55    return 0;
56  }
```

Listing A.1: Zeta Generation in Plantard Domain

A.2. Generated Output: zetas[128]

```
const int16_t zetas[128] = {
        -1353, 950, 1381, 856, 720, -166, 18, 1161,
        1242, 213, -1467, 255, -1030, 145, -858, 1252,
        -1008, 1564, -691, 372, 64, 799, -1330, 769,
        1442, -203, -1462, -1087, 1388, -357, -964, 1073,
        -1524, 1572, -371, 1038, -1647, -1368, -1456, -700,
        -594, 1635, -1180, 457, -1389, -1372, -1616, -1033,
        1640, -748, 41, 980, -1478, 1210, -1535, -802,
        613, 1255, 265, -1217, 1652, 26, 1040, 500,
        302, -495, 174, 1236, -1076, 507, 306, -237,
        1140, 292, -1636, 1006, -865, -864, -1270, 1310,
        -491, -44, 1569, -334, 1088, 267, 693, -243,
        1211, -122, -1551, 1495, 293, 589, 257, 1596,
        724, 92, 351, 1001, -1367, 47, -1449, 1416,
        -111, 1163, -86, 1111, -310, -21, -840, -916,
        1248, 600, 697, 15, 1506, 596, 537, -318,
        434, 1361, 1176, -715, 1452, 442, 1035, -1487
};
```