Radboud University

Master's Thesis Computing Science

In Cyber Security

KYBER on RISC-V

Supervisor:
dr. Peter Schwabe

Second Assessor:
prof. dr. Lejla Batina

Author:
Denisa GRECONICI

Radboud University Nijmegen
Institute for Computing and Information Sciences
Digital Security

January 3, 2020

Abstract

In this thesis we present a speed optimization for Kyber on the recently developed open-source RISC-V architecture. Kyber is a key-encapsulation mechanism based on module-LWE and it is currently competing in round 2 of the NIST competition. Our speed optimization focuses on the Number Theoretic Transform (NTT) and its inverse (INTT) for round 1 and round 2 of Kyber, functions that are important for performing fast polynomial multiplication. We also optimize the Montgomery and Barrett reductions inside the NTT and INTT in Kyber. The large number of registers in RISC-V allow us to merge up to 4 levels of the NTT and INTT and load 16 polynomial coefficients at a time, significantly reducing the number of loads and stores and implicitly the cycle count. For round 2 we also use instruction interleaving as optimization technique, improving the speed even more. For round 1 of Kyber we obtain an NTT of 27390 cycles and an INTT of 25669 cycles. For round 2 of Kyber our NTT is 14348 cycles and the INTT is 13742 cycles. Our results are around 70–80 % faster than the reference implementation of Kyber on RISC-V, although it should be noted that the reference implementation was not optimized for speed in any way.

Contents

List of Figures

List of Algorithms

List of Code Listings

List of Tables

Abbreviations

AES	Advanced Encryption Standard
CBD	Centered Binomial Distribution
CCA2	Adaptive Chosen Ciphertext Attack
CPA	Chosen Plaintext Attack
CT	Cooley-Tukey Butterfly
DFT	Discrete Fourier Transform
DH	Diffie-Hellman
DLP	Discrete Logarithm Problem
ECDSA	Elliptic Curve Digital Signature Algorithm
FFT	Fast Fourier Transform
FO	Fujisaki-Okamoto Transform
GS	Gentleman-Sande Butterfly
IND	Indistinguishable
INTT	Inverse Number Theoretic Transform
ISA	Instruction Set Architecture
KEM	Key Encapsulation Mechanism
LWE	Learning With Errors
MAC	Message Authentication Code
NIST	National Institute of Standards and Technology
NTT	Number Theoretic Transform

PKE	Public Key Encryption
PRF	Pseudo-Random Function
RISC	Reduced Instruction Set Computer
SVP	Shortest Vector Problem
XOF	Extendable Output Function

1. Introduction

1.1 Context

Current cryptographic applications combine both symmetric cryptography and asymmetric cryptography in order to preserve security goals like confidentiality, integrity and availability. For two parties A and B that want to encrypt messages to each other, confidentiality means that only A and B can decrypt the messages. Integrity implies that no third party (adversary/attacker) can alter the messages and availability means that no one can prevent the communication between A and B. We will focus on the first two security goals as cryptography by itself cannot guarantee availability.

In general, symmetric cryptography is used to encrypt and decrypt messages between two parties using a shared secret key. Another shared secret key is usually used to compute the Message Authentication Code (MAC) which is a unique information per message that ensures its integrity. The most used algorithm for symmetric encryption is the Advanced Encryption Standard (AES) [18]. If an adversary learns the secret keys, confidentiality and integrity are broken (i.e. he can encrypt/decrypt any message, modify it and compute the correct MAC). Intuitively, a way of preventing the adversary from learning the secret keys is to make sure that the keys are exchanged on a secure channel and that they are updated often. In this way we would reduce the chances of the attacker to guess the keys. The previously-mentioned objectives are implemented nowadays through asymmetric (or public key) cryptography. It consists of algorithms that derive a secret key shared between two parties (e.g. Diffie-Hellman (DH) [16], Elliptic Curve DH), algorithms that encrypt a key previously chosen by the initiating party (e.g. RSA [34]) and signing algorithms that provide integrity (e.g. RSA, Elliptic Curve Digital Signature Algorithm (ECDSA)). The idea is to use asymmetric cryptography to establish the shared key and then use that key to symmetrically encrypt and authenticate messages. In public-key cryptography each party has a pair of keys: a private (secret) key and a public key. One uses someone's public key to encrypt messages to them and the secret key to decrypt messages he receives. In key-exchange algorithms, two parties send their public key to each other and each of them combines his own secret key with the received public key and derive the shared key. In order to sign a message one uses his secret key and everyone else can verify the signature validity by using the corresponding public key.

1.2 Motivation

Most of the public-key cryptography used today is based on the hardness of solving the discrete-logarithm problem (DLP) and factoring integers. In 1994 Peter Shor published a paper showing how to solve the DLP and factor integers in polynomial time using quantum algorithms [36]. This type of algorithms run on quantum computers, machines based on quantum mechanics that could solve problems that a classical compute cannot. While classical computers work with bits which take the values 0 or 1, quantum computers are based on qbits which can take the values 0, 1 or both of them in the same time. This thesis will not discuss in any way the working principles of a quantum computer. For those looking for an introduction into the topic, please look at [29].

Big companies and research institutes are already researching the topic of creating a quantum computer for quite some time. In fact, this year Google developed a quantum computer that solves in a fraction of time a very specific problem (not cryptographic related) which would not be feasible on the most advanced classical computers nowadays [3]. It is fair to assume that with the amount of research going on in the area of quantum computers, there are high chances they become a reality within a few decades. This implies that the earlier introduced algorithm for factoring ans solving DLPs by Shor is applicable, breaking all public key cryptography that we use today. In response to this threat, the National Institute of Standards and Technology (NIST) started a competition in 2016 to encourage researchers to propose asymmetric cryptographic schemes that would be resistant to quantum computers[1]. This category of asymmetric schemes is also known as post-quantum cryp-

[1]`https://csrc.nist.gov/News/2016/Public-Key-Post-Quantum-Cryptographic-Algorithms`

tography and together with the symmetric cryptography used with larger keys (e.g. AES, hashing algorithms) are assumed to be secure against quantum computers. Initially, 69 candidates qualified for round 1 of the NIST competition, and since January 2019, 26 candidates are competing in round 2. The post-quantum schemes that are currently in the NIST competition[2] are either key encapsulation mechanisms (KEMs) or signatures and they can be classified into the following categories[3]:

- **Hash-based cryptography** comprises of hash-based signatures that rely only on certain properties of the underlying hash functions like second-preimage resistance. In round 2 of the NIST competition there are only 2 hash-based signatures: SPHINCS+[4] and Picnic[5]. To be noted that Picnic is not only based on hash functions, but it requires a zero-knowledge proof system based on hash functions and block ciphers. However out of these 5 categories, it fits best in the hash-based cryptography.
- **Code-based cryptography** is based on the hardness of decoding a codeword with random errors. In the NIST competition there are currently 7 code-based schemes, all of them KEMs. For example, one of them is Classic McEliece[6] which is a representative scheme in the field as it is based on the first code-based scheme which uses Goppa codes (McEliece from 1978 [27]).
- **Multivariate cryptography** is based on the hardness of solving systems of quadratic equations in many variables. In the NIST competition there are currently 4 multivariate-based schemes, all of them signature schemes (e.g. MQDSS[7]).
- **Isogeny-based cryptography** is based on the hardness of finding isogenies (mappings) between elliptic curves over finite fields. There is only one KEM scheme currently in the NIST competition, namely SIKE[8].
- **Lattice-based cryptography** includes KEMs and signature schemes based on NTRU [21] and on the Learning With Errors (LWE) problem

[2] `https://csrc.nist.gov/projects/post-quantum-cryptography/round-2-submissions`
[3] `https://pqc-wiki.fau.edu/`
[4] `https://sphincs.org/`
[5] `https://microsoft.github.io/Picnic/`
[6] `https://classic.mceliece.org/`
[7] `http://mqdss.org/`
[8] `https://sike.org`

[32]. The security of these systems is reduced from the hardness of the lattice problems such as finding the shortest vector in a lattice (SVP problem). In the NIST competition there are currently 12 lattice-based schemes, 9 KEMs (e.g. Kyber[9], NewHope[10], and NTRU Prime[11]) and 3 signature schemes (Dilithium [12], FALCON[13], and qTESLA[14]).

In this thesis we will focus on Kyber, a module-LWE-based key-encapsulation mechanism (see the definitions in Section 2.3) that is currently in round 2 of the NIST post-quantum competition. Because performance is one of the important criteria for NIST, we aim to optimize both rounds of this scheme for speed on the RISC-V architecture. Kyber [12] and NewHope [1] are the only two key-encapsulation schemes in the NIST competition that use in their definition the so called NTT – Number Theoretic Transform (defined in Section 2.2) to compute fast polynomial multiplication. Our speed optimization techniques will target the NTT implementation and its inverse operation (INTT) as they are the second most time consuming functions in Kyber after Keccak [4, Section 2.1]. We chose RISC-V [15] because it is a promising new open source instruction set architecture which attracted a community of more than 325 members that are actively developing it. It started as a university project at University of California, Berkeley, and now big players such as Google, NVIDIA and NXP are investing in it.

Thesis Structure. In the following lines we will give the outline for the rest of the thesis. The second chapter introduces the notation and theoretical aspects the reader needs to know in order to understand how Kyber works. In the third chapter we discuss efficient implementations for Kyber, more specifically how to efficiently compute polynomial multiplication using fast Fourier transform algorithms (the NTT). Chapter 5 presents our RISC-V speed optimization techniques for Kyber. In Chapter 6 we illustrate the results we obtained. Finally, Chapter 7 captures our conclusion.

[9]`https://pq-crystals.org/kyber/index.shtml`
[10]`https://newhopecrypto.org/`
[11]`https://ntruprime.cr.yp.to/`
[12]`https://pq-crystals.org/dilithium/index.shtml`
[13]`https://falcon-sign.info/`
[14]`https://qtesla.org/`
[15]`https://riscv.org/risc-v-foundation/`

2. Preliminaries

This chapter contains the theoretical background the reader needs to have in order to understand the basic concepts of Kyber. The notation and definitions used in this thesis are also covered here.

2.1 Notation and Definitions

The Set of Integers Modulo q. In the next three paragraphs we stick to the notation given in the Kyber specifications document [4, Section 1.1]. We denote by $\mathbb{Z}$ the set of integer numbers. For a prime number q, we refer to $\mathbb{Z}_q^{\pm}$ as the set of integers modulo q (mod q) within the range $[-q/2, q/2)$ and to $\mathbb{Z}_q^{+}$ the set of integers modulo q within the range $[0, q)$. When the representation does not matter, we simply write this set as $\mathbb{Z}_q$.

Byte Array Concatenation. We denote by $a||b$ the concatenation of two byte arrays a and b.

Sampling. Given the relationship $x \leftarrow S$ we say that x is chosen uniformly at random from the set S. If S is replaced by a probabilistic distribution D, we say that x is sampled according to D. We denote by $x \leftarrow D(y)$ a deterministic sampling from the distribution D with the seed y. This means that for the same input y, the output x will always get the same value, which is uniformly distributed over D.

Centered Binomial Distribution. For a positive integer η, we define the centered binomial distribution CBD $= \beta_\eta$ as: *sample* $(a_1, ...a_\eta, b_1...b_\eta) \leftarrow \{0,1\}^{2\eta}$ and output $\sum_{i=1}^{\eta}(a_i - b_i)$ [4].

Vectors. Vectors are represented with small bold letters (i.e. $\mathbf{v}$) and by default they are interpreted as column vectors. The matrices are denoted by capital bold letters (i.e. $\mathbf{A}$). By $\mathbf{A}^T$ and $\mathbf{v}^T$ we mean the transpose of $\mathbf{A}$ and $\mathbf{v}$. We denote by $\mathbf{v}[i]$ the i'th element of the vector $\mathbf{v}$ and with $\mathbf{A}[i][j]$ the element form the i'th row and j'th column of the matrix $\mathbf{A}$.

For two vectors of the same length we define the inner product as $\langle \mathbf{a}, \mathbf{b} \rangle = \sum_i a_i b_i$, which is is also equivalent with the product $\mathbf{a}^T \cdot \mathbf{b}$. Following the same logic, for a matrix $\mathbf{A}$ and a vector $\mathbf{b}$, $\langle \mathbf{A}, \mathbf{b} \rangle = \mathbf{A}^T \cdot \mathbf{b}$.

Polynomials. A polynomial p of degree $n \neq 0$ is defined as $p = \sum_{i=0}^{n} p_i X^i = p_0 + p_1 X + ... + p_n X^n$. It can also be represented by a n-dimensional vector $\mathbf{v}$ where $v[i] = p_i$. We denote by $\mathbb{Z}[X]$ the set of polynomials with coefficients in $\mathbb{Z}$.

A special collection of polynomials is represented by the ring $R_q = \mathbb{Z}_q[X]/(X^n + 1)$ which consists of all polynomials with degree less than n, and with coefficients integers modulo q. Reductions of polynomials to R_q follow the rule $X^n \equiv -1 \pmod{X^n + 1}$. In this thesis we assume that n is a power of two (i.e. $n = 2^m, m > 0$) and $q \equiv 1 \pmod{2n}$.

In general, when multiplying polynomials of degree n, the result can have a maximum degree of $2n$. However, in this thesis polynomial multiplication is done in R_q and we assume that the result is automatically reduced to R_q. An explanation for this assumption will be given in the definition of the Number Theoretic Transform.

2.2 The Number Theoretic Transform

The Number Theoretic Transform (NTT) is based on the Discrete Fourier Transform (DFT) and it can be used to compute fast polynomial multiplication in R_q. Unlike the DFT which operates on complex numbers, the NTT is applied on integers in $\mathbb{Z}_q$.

As explained in [9], an n-degree polynomial in R_q can be represented either by its n coefficients, or by n values resulting from evaluating the polynomial at the points $\omega_n^0, \omega_n^1 ... \omega_n^{n-1}$. Here, ω_n is a primitive n'th root of unity meaning that $\omega_n^n = 1 \mod q$ and for $1 \leq k < n$, $\omega_n^k \neq 1 \mod q$. In order for the primitive n'th roots of unity to exist, n must divide $q-1$. A forward NTT can be seen as a mapping from the coefficient representation of a polynomial to the values obtained by evaluating the polynomial at the powers of the n'th primitive root of unity. The reverse mapping is done through the inverse NTT (here denoted INTT) and is also called interpolation.

For a polynomial $g \in R_q, g = \sum_{i=0}^{n-1} g_i X^i$, the n'th root of unity ω and the $2n$'th root of unity $\psi = \sqrt{\omega}$ we define the NTT and INTT as:

$$\hat{g} = NTT(g) = \sum_{i=0}^{n-1} \hat{g}_i X^i, \qquad with \qquad \hat{g}_i = \sum_{j=0}^{n-1} \psi^j g_j \omega^{ij}; \qquad and$$

$$g = INTT(\hat{g}) = \sum_{i=0}^{n-1} g_i X^i, \qquad with \qquad g_i = n^{-1}\psi^{-i} \sum_{j=0}^{n-1} \hat{g}_j \omega^{-ij}.$$

With the above-mentioned definitions, the product between two polynomials $a, b \in R_q$ can be computed efficiently as: a $\cdot$ b = INTT(NTT(a) $\circ$ NTT(b)), where $\circ$ represents the coefficient-wise product (also known as convolution).

Usually the definition of the NTT and INTT does not contain the multiplication by the factors ψ and ψ^{-1}. However, with this modification we are able to directly compute polynomial multiplication with coefficients in $\mathbb{Z}_q[X]$, modulo $X^n + 1$. This technique is called negative wrapped convolution and as a consequence of using it, we need to ensure that the $2n$'th root of unity ψ exists. Therefore $2n$ must divide $q - 1$.

In this thesis, the values that are in the NTT domain will be marked with a hat (i.e. $\hat{\mathbf{A}}$). When we transform a vector or a matrix of polynomials to/from the NTT domain, we apply the NTT/INTT operation to each individual element of the vector/matrix. The number of NTT/INTT operations will be equal to the dimension of the vector/matrix.

An important property of NTT and INTT is that they are linear, so in other words (I)NTT(a)+(I)NTT(b) = (I)NTT(a+b). When we want to multiply a polynomial matrix $\mathbf{A} \in R_q^{k \times k}$ with a polynomial vector $\mathbf{v} \in R_q^k$, we end up with sums of polynomial products. Naturally, we would apply the rule a $\cdot$ b =INTT(NTT(a) $\circ$ NTT(b)) for each polynomial multiplication and then add them up to compute the result vector. However, due to the linear property of the INTT, we can first add the coefficient-wise products and apply the INTT only at the end. In this way, instead of computing k^2 INTT's, we only compute k which is the dimension of the result vector.

2.3 Cryptographic Background

For all the following definitions in this section we kept their initial description from their references. However, according to [2] the secret **s** (or the coefficients of the secret **s**) does not have to be chosen uniformly at random, but the problems remains hard if **s** is chosen from the same distribution as the error **e** (or the coefficients of error **e**). In Kyber the secret is chosen from the same distribution as the error.

2.3.1 Learning With Errors

The Learning With Errors (LWE) problem was introduced by Regev in [32] and it falls under lattice-based cryptography asymptotically. This is because its security is proven to be as hard as the worst-case lattice problems. The LWE problem can be seen as solving a linear system in $\mathbb{Z}_q$ to which an unknown vector of small errors is added. Without the errors, the system could easily be solved by Gausian Elimination, but the existence of the unknown errors makes it hard to solve. The LWE-based cryptographic systems usually require a key size of n^2 as the key consists of n vector samples $\mathbf{a_1}, \mathbf{a_2}...\mathbf{a_n} \in \mathbb{Z}_q^n$ [33]. More formally, the LWE is defined as follows:

Definition 1. For a secret vector **s** and a vector **a** both chosen uniformly at random from $\mathbb{Z}_q^n$, and for an error $e \in \mathbb{Z}_q$ sampled from a distribution D, the following problems are defined:

Search LWE. Given m samples of the form $(\mathbf{a}, \langle \mathbf{a}, \mathbf{s} \rangle + e) \in \mathbb{Z}_q^n \times \mathbb{Z}_q$, recover the secret vector **s**.

Decisional LWE. Given m samples of the form $(\mathbf{a}, b) \in \mathbb{Z}_q^n \times \mathbb{Z}_q$, decide if all samples are of form $(\mathbf{a}, \langle \mathbf{a}, \mathbf{s} \rangle + e)$ or if they are sampled from the uniform distribution over $\mathbb{Z}_q^n \times \mathbb{Z}_q$.

2.3.2 Ring-LWE

The ring-LWE problem was introduced in [25] and it is similar to the LWE problem, but now the vectors in $\mathbb{Z}_q^n$ are replaced by polynomials in R_q with degree less than n. For cryptographic systems this means that the key size is of order n instead of n^2, as now the n vectors are sampled from R_q rather than from $\mathbb{Z}_q^n$. Another advantage over LWE is that for certain parameter choices fast Fourier-based algorithms can be applied to compute the multiplication in R_q.

The idea behind ring-LWE is to give some structure to the LWE system by sampling only one vector $\mathbf{a_1} = (x_1, x_2...x_n) \in \mathbb{Z}_q^n$ [33]. The rest of $n-1$ vectors would be obtained by the rule $\mathbf{a_i} = (x_i, ..., x_n, -x_1, ..., -x_{i-1})$, therefore only the first vector $\mathbf{a_1}$ is needed to represent the structure. This new representation can be converted to the ring R_q. The security of ring-LWE is proven to be as hard as the worst-case ideal lattice problems asymptotically [25]. The definition of ring-LWE is given as follows:

Definition 2. For a fixed secret ring element $\mathbf{s}$ and the ring element $\mathbf{a}$, both with coefficients chosen uniformly at random from $\mathbb{Z}_q$, and for an error $e \in R_q$ with its coefficients sampled from a distribution D, the following problems are defined:

Search Ring-LWE. Given m samples of the form $(\mathbf{a}, \mathbf{a} \cdot \mathbf{s} + e) \in R_q \times R_q$, recover the secret $\mathbf{s}$.

Decisional Ring-LWE. Given m samples of the form $(\mathbf{a}, b) \in R_q \times R_q$, decide if all samples are of form $(\mathbf{a}, \mathbf{a} \cdot \mathbf{s} + e)$ or if they are sampled from the uniform distribution over $R_q \times R_q$.

2.3.3 Module-LWE

Module-LWE was introduced in [24]. It is very similar to ring-LWE, but instead of sampling one polynomial in R_k, we now sample a vector of k polynomials in R_k. We call this vector a module, and we say that k is the rank of the module. If we fix k to 1, the problem becomes ring-LWE and if we choose $k = n$. The hardness of module-LWE is compared with the the hardness of solving module lattices problems. For module-LWE, the following definition is given:

Definition 3. For a secret module element $\mathbf{s} \in R_q^k$ and the module element $\mathbf{a} \in R_q^k$, with the coefficients of each polynomial in R_q chosen uniformly at random from $\mathbb{Z}_q$, and for an error $e \in R_q$ with its coefficients sampled from a distribution D, the following problems are defined:

Search Module-LWE. Given m samples of the form $(\mathbf{a}, \mathbf{a} \cdot \mathbf{s} + e) \in R_q^k \times R_q$, recover the secret $\mathbf{s}$.

Decisional Module-LWE. Given m samples of the form $(\mathbf{a}, b) \in R_q^k \times R_q$, decide if all samples are of form $(\mathbf{a}, \mathbf{a} \cdot \mathbf{s} + e)$ or if they are sampled from the uniform distribution over $R_q^k \times R_q$.

2.4 Public Key Encryption Scheme.

A Public Key Encryption scheme (PKE) consists of three probabilistic algorithms (KeyGen, Enc, Dec) and a message space M [12].

1. **KeyGen()** (Key Generation) is an algorithm that outputs a key pair (pk, sk) where pk is the public key and sk is the corresponding private key.
2. **Enc(pk)** (Encryption) is a probabilistic algorithm that takes a public key pk and a message $m \in M$ and produces a ciphertext c.
3. **Dec(sk, c)** (Decryption) is a deterministic algorithm that takes as input a secret key sk and a ciphertext c and return a message $m \in M$, or in case of rejection, the symbol $\perp$.

2.5 Key Encapsulation Mechanisms.

A Key Encapsulation Mechanism (KEM) consists of three algorithms (KeyGen, Encaps, Decaps) and a key space K [12].

1. **KeyGen()** (Key Generation) is a probabilistic algorithm that outputs a key pair (pk, sk) where pk is the public key and sk is the corresponding private key.
2. **Encaps(pk)** (Encapsulation) is a probabilistic algorithm that takes a public key and outputs a ciphertext c and a key $k \in K$.
3. **Decaps(sk, c)** (Decapsulation)is a deterministic algorithm that takes as input a secret key sk and a ciphertext c and return a key $k \in K$, or in case of rejection, the symbol $\perp$.

3. Kyber

3.1 Round 1

Kyber [12, 4] is a key encapsulation mechanism based on the hardness of solving the Module-LWE problem. This implies that the arithmetic operations are performed over the polynomial ring R_q. For efficiency reasons, Kyber uses the NTT to perform polynomial multiplications. With this in mind, the modulus q, and the parameter n are fixed to $q = 7681$ and $n = 256$ such that the primitive n'th and $2n$'th roots of unity exist (the required condition for the NTT to exist). The resulting ring is $R_{7681} = \mathbb{Z}_{7681}[X]/(X^{256} + 1)$ and the inverse of n modulo q is 7651. With these parameters defined, the 256'th primitive root of unity is fixed to $\omega = 3844$ and its inverse modulo q is $\omega^{-1} = 6584$. Accordingly, the 512'th primitive root of unity is $\psi = 62$ and its inverse is $\psi^{-1} = 1115\ mod\ q$. Depending on the desired security level, Kyber comes in three versions: Kyber-512 comparable with the security of AES-128, Kyber-768 comparable with the security of AES-192 and Kyber-1024 comparable with the security of AES-256. The security level is defined by the parameter k which represents the module rank (from Module-LWE), and it can take the values 2, 3 and 4. By default, k is set to 3.

The next two sections will describe a simplified version of the Kyber scheme. For more details, please have a look at [4, Ch 1.2,Ch1.3]. In the Algorithms 1-6 the pseudo-random function (PRF) is instantiated with SHAKE-256, the extendable output function (XOF) with SHAKE-128 and the hash functions H and G with SHA3-256, respectively with SHA3-512. These functions belong to the FIPS-202 standard [17]. PRF and XOF take as input a random value and a nonce. The latter is implemented as a counter starting from 0, but for simplicity it is omitted in the algorithm descriptions. The Kyber KEM is build up in two stages. The first step is to construct a public key encryption scheme resistant against a passive attacker, namely an IND-CPA-secure scheme (Indistinguishable under Chosen Plaintext Attack). The second step is to make Kyber resistant to an active attacker by applying a slightly modified version of the Fujisaki-Okamoto (FO) transform [19] and obtain a key encapsulation mechanism which is IND-CCA2-secure (Indistinguishable under Adaptive Chosen Ciphertext Attack). More details about IND-CPA and IND-CCA2 can be found in [8].

3.1.1 IND-CPA Scheme

CPA Key Generation. The CPA Key Generation of Kyber is illustrated in Algorithm 1 and it produces as output a pair consisting of a public key and a secret key. The secret key $\mathbf{s}$ is a vector of k polynomials in R_q, each coefficient being deterministically sampled from the CBD distribution. The vector is transformed and stored in the NTT domain, becoming $\hat{\mathbf{s}}$.

The public key is the pair $(\mathbf{t}, \rho)$ where $\mathbf{t} = \mathbf{A}\mathbf{s} + \mathbf{e}$ is a vector of k polynomials and ρ is the public seed from which the matrix $\mathbf{A}$ is generated. The public information $\mathbf{A}$ is a $k \times k$ matrix of polynomials in R_q. Since $\mathbf{A}$ is only used in the NTT form, the designers of Kyber have chosen to save k^2 transformations and sample it directly in the NTT domain from a uniformly random distribution. This is possible because the NTT maps uniformly random coefficients to uniformly random coefficients [4]. As a consequence, the NTT explicitly appears in the scheme to clarify when a transformation is needed.

Algorithm 1: CPA.KeyGen()

```
  Output: Public key $pk := (\mathbf{t}, \rho)$
  Output: Secret key: $sk := \hat{\mathbf{s}}$
1 $\rho, \sigma \leftarrow \{0,1\}^{256} \times \{0,1\}^{256}$
2 $\hat{\mathbf{A}} \in R_q^{k \times k} \leftarrow U(XOF(\rho))$
3 $\mathbf{s} \in R_q^k \leftarrow CBD(PRF(\sigma))$
4 $\mathbf{e} \in R_q^k \leftarrow CBD(PRF(\sigma))$
5 $\hat{\mathbf{s}} := \mathbf{NTT}(\mathbf{s})$
6 $\mathbf{t} := \mathbf{NTT^{-1}}(\hat{\mathbf{A}} \circ \hat{\mathbf{s}}) + \mathbf{e}$
7 $sk := \hat{\mathbf{s}}$
8 $pk := (Compress(\mathbf{t}), \rho)$
9 return $(pk, sk)$
```

Another design decision in Kyber was to compress the $\mathbf{t}$ part of the public key by dropping some least significant bits without affecting the correctness of the scheme. This is done through the function Compress (line 8, Algorithm 1).

Because in this thesis we focus on reducing the cost of the NTT and INTT, we want to compute how many times these transformations are used in Kyber. As they are already part of the scheme, it feels natural to count them while describing each algorithm. Therefore, in Algorithm 1, line 5 we have k NTT transformations as the vector $\mathbf{s}$ contains k polynomials. At line 6, $\hat{\mathbf{s}}$ and $\hat{\mathbf{A}}$ are already in the NTT domain, so no NTT operation is needed. As for the INTT, it is applied on a product resulting in a vector of k polynomials, thus we have k INTT operations.

CPA Encryption. The key encryption algorithm is described in Algorithm 2. This function takes as input a public key, the message to be encrypted and a random coin. The message to be encrypted has a fixed size of 256 bits and it can be represented as a polynomial in R_q with coefficients in $\{0,1\}$ (i.e., each bit is represented by a coefficient).

Algorithm 2: CPA.Encrypt(pk,m,μ)

```
   Input  : Public key $pk := (\mathbf{t}\rho)$
   Input  : Message $m \in R_q$
   Input  : Random coins $\mu \in \{0,1\}^{256}$
   Output: Ciphertext $c := (\mathbf{u}, v)$,
1  $\rho \leftarrow \{0,1\}^{256}$
2  $t := Decompress(t)$
3  $\hat{\mathbf{A}} \in R_q^{k\times k} \leftarrow U(XOF(\rho))$
4  $\mathbf{r} \in R_q^k \leftarrow CBD(PRF(\mu))$
5  $\mathbf{e_1} \in R_q^k \leftarrow CBD(PRF(\mu))$
6  $e_2 \in R_q \leftarrow CBD(PRF(\mu))$
7  $\hat{\mathbf{r}} := \mathbf{NTT}(\mathbf{r})$
8  $\mathbf{u} := \mathbf{NTT^{-1}}(\hat{\mathbf{A}}^{\mathbf{T}} \circ \hat{\mathbf{r}}) + \mathbf{e_1}$
9  $m := \lceil q/2 \rfloor \cdot m$
10 $v := \mathbf{NTT^{-1}}(\mathbf{NTT}(\mathbf{t})^{\mathbf{T}} \circ \hat{\mathbf{r}}) + \mathbf{e_2} + m$
11 return $(Compress(\mathbf{u}), Compress(v))$
```

After $\mathbf{t}$ is decompressed, the public information is generated from a uniform distribution using the seed ρ. To be noted that $\hat{A}$ will be the same as the one in the key generation phase because it is generated from the same seed. In lines 4, 5 and 6 the ephemeral secret key $\mathbf{r} \in R_q^k$ together with the errors $\mathbf{e_1} \in R_q^k$ and $\mathbf{e_2} \in R_q$ are deterministically generated from CBD using the input seed μ. In line 9 an error tolerance is created by mapping the coefficients of the message from 0 to 0 and from 1 to $\lceil q/2 \rfloor$, where $\lceil x \rfloor$ represents the closest integer to x, ties being rounded up. In this context, this operation is identified with the LWE error correction. The output ciphertext is of form $(\mathbf{u}, v)$ where $\mathbf{u}$ and v are defined in line 8 and 10.

For Algorithm 2 we have k NTT transformations at line 7 as the vector $\mathbf{r}$ contains k polynomials. Another k NTT transformations are needed at line 10 for the vector $\mathbf{t}$. In total, $2k$ NTT's are used for this algorithm. As for the INTT we have k transformations at line 8, and 1 at line 10, resulting in $k + 1$ INTT's.

CPA Decryption. Algorithm 3 illustrates the decryption. The Decompress() function restores the initial size of the ciphertext values $\mathbf{u}$ and v. The message is recovered from the equation $m = v - \mathbf{s^T} \cdot \mathbf{u}$. For decryption correctness we refer to [12, Ch. 3]. In line 4, each coefficient of m is decrypted to 0 if its value is closer to 0 or to 1 if its value is closer to $\lceil q/2 \rfloor$.

Algorithm 3: CPA.Decrypt(sk,c)

```
  Input  : Secret key: $sk := \hat{\mathbf{s}}$
  Input  : Ciphertext $c := (\mathbf{u}, v)$
  Output: Message m $\in R_q$
1 $\mathbf{u} := Decompress(\mathbf{u})$
2 $v := Decompress(v)$
3 $m := v - \mathbf{NTT^{-1}}(\hat{\mathbf{s}}^{\mathbf{T}} \circ \mathbf{NTT}(\mathbf{u}))$
4 return $\lceil 2/q \rfloor \cdot m$
```

In Algorithm 3, as $\mathbf{u}$ is a vector of k polynomials, there are k NTT operations needed to transform $\mathbf{u}$ into the NTT domain. The secret $\hat{\mathbf{s}}$ is already saved in the NTT domain, so no operation is needed. The result of $(\hat{\mathbf{s}}^{\mathbf{T}} \circ \mathbf{u})$ is one polynomial, therefore only 1 INTT operation is needed.

3.1.2 IND-CCA2 Scheme

Before explaining the IND-CCA2 scheme illustrated in Algorithm 4, 5 and 6, we will give an intuition on why this step is needed. For simplicity we will call Alice the one that encrypts/encapsulates a message and Bob the one that decrypts/decapsulates it. In the previous setting it is assumed that

Alice samples the errors $\mathbf{e_1}, e_2$ and the ephemeral secret key $\mathbf{r}$ from CBD. However, an attacker can cheat and sample these values in such a way that he could learn something about the secret key [10, 22, 7, 31]. That is why in the IND-CCA2 step Alice is forced to deterministically derive the values $\mathbf{e1}, e2$ and $\mathbf{r}$ from the hash of the message to be encapsulated and the hash of the public key of Bob. After decrypting the ciphertext, Bob can generate the same values $\mathbf{e1}, e2$ and $\mathbf{r}$. With this information he will re-encapsulate the message he obtained and verify that the ciphertext that he received was fairly generated by Alice.

CCA KeyGeneration. In Algorithm 4, the public key pk and the pre-secret key sk' are obtained by calling CPA.KeyGen(). The output of this function will be the public key pk and the secret key sk, which is the concatenation of the following values: sk', the public key pk, the hash of the public key $H(pk)$ and a random 256-bit value z. The hash of the public key will be used in the decapsulation step and it is appended to the secret key so that it is not recomputed every time.

Algorithm 4: CCA.KeyGen()

```
  Output: Public key $pk := (\mathbf{t}, \rho)$
  Output: Secret key: $sk := (sk'||pk||H(pk)||z)$
1 $z \leftarrow \{0,1\}^{256}$
2 $(pk, sk') := CPA.KeyGen()$
3 $sk = (sk'||pk||H(pk)||z)$
4 return $(pk, sk)$
```

CCA Key Encapsulation. Algorithm 5 ensures that Alice and Bob can deterministically generate the same ciphertext, given the message m and some shared randomness μ. The first step is to generate the random message m and then derive the seed μ, together with a temporary key $\bar{K}$ by hashing the hash of the message to be encrypted and the hash of the public key. The ciphertext c is obtained by applying CPA.KeyEncaps(pk,m,μ), ensuring that Alice will deterministically derive the values $\mathbf{e1}, e2$ and $\mathbf{r}$ from the seed μ .

The shared key K that will be used for symmetric encryption between Alice and Bob is obtained by hashing the temporary key $\bar{K}$ together with the hash of the ciphertext c.

CCA Key Decapsulation . The CCA decapsulation mechanism is presented in Algorithm 6. First, Bob decrypts the ciphertext into the message

Algorithm 5: CCA.Encapsulation(pk)

```
Input  : Public key $pk := (\mathbf{t}\rho)$
Output: Ciphertext $c := (\mathbf{u}, v)$,
Output: Shared key $K \in \{0,1\}^{256}$
1 $m \leftarrow \{0,1\}^{256}$
2 $m := H(m)$
3 $(\bar{K}, \mu) := G(m||H(pk))$
4 $c := CPA.Encrypt(pk, m, \mu)$
5 $K := H(\bar{K}||H(c))$
6 return $(c, K)$
```

m' using the CPA decryption function. Having m' and knowing the hash of his public key, Bob can derive the temporary key $\bar{K}'$ and the shared seed μ'. He will now re-encrypt the message m' using the CPA.KeyEnc function and the seed μ'. If the ciphertext that Bob computed matches the ciphertext that he received from Alice, he will accept the encapsulation and he will compute the shared key K as the hash of the temporary key $\bar{K}'$ and the hash of the ciphertext c. Otherwise, he will compute a random key K using the random value z as shown in line 7.

Algorithm 6: CCA.Decapsulation(sk,c)

```
Input  : Secret key: $sk := (sk'||pk||h||z)$
Input  : Ciphertext $c$
Output: Shared key $k \in \{0,1\}^{256}$
1 $m' := CPA.Deccrypt(sk', c)$
2 $(\bar{K}', \mu') := G(m', h)$
3 $c' := CPA.Encrypt(pk, m', \mu')$
4 if $c = c'$ then
5     return $K := H(\bar{K}'||H(c))$
6 else
7     return $K := H(z||H(c))$
8 end
9 return $K$
```

3.2 Round 2

While working on this thesis, a new version of Kyber was released as D'Anvers pointed out that the security proof of Kyber stands only if the public key is not compressed [1]. Therefore, in the second round the public key is preserved in its natural length, but this time it is represented and transmitted in the NTT domain. To compensate for not compressing the public key, besides other changes [5, p2], the prime modulus q was updated from 7681 to 3329 resulting into the ring $R_{3329} = \mathbb{Z}_{3329}[X]/(X^{256} + 1)$.

As a consequence of choosing this q, there are no $2n$'th primitive roots of unity for $n = 256$. However, [26] shows that we can still compute the NTT with this configuration. In round 1, the NTT was represented by 256 coefficients that could be seen each as polynomials of degree 0. With the parameters from round 2, the NTT will be interpreted as 128 coefficients, each being a polynomial of degree 1. Therefore, for a polynomial $f \in R_q$ the new definition of the NTT in Kyber round 2 is [5]:

$$NTT(f) = \hat{f} = (\hat{f}_0 + \hat{f}_1 X, \hat{f}_2 + \hat{f}_3 X, ...\hat{f}_{254} + \hat{f}_{255} X),$$

where

$$\hat{f}_{2i} = \sum_{j=0}^{127} f_{2j} \zeta^{(2br_7(i)+1)j}$$

$$\hat{f}_{2i+1} = \sum_{j=0}^{127} f_{2j+1} \zeta^{(2br_7(i)+1)j}$$

Here, the 256'th primitive root of unity is $\zeta = 17$ and br7(x) reverses the 7-bit representation of x. As coefficients are now polynomials of degree 1, the coefficient-wise operation $\circ$ will be redefined as multiplication of degree 1 polynomials modulo $X^2 - \zeta^{2br_7(i)+1}$.

[1] `https://csrc.nist.gov/CSRC/media/Projects/Post-Quantum-Cryptography/documents/round-1/official-comments/CRYSTALS-KYBER-official-comment.pdf`

4. Efficient NTT in Kyber

4.1 NTT Implementations

The most expensive functions used in Kyber are the ones based on the Keccak permutation [11] (SHA3-256, SHA3-512, SHAKE-128, SHAKE-256) and the ones involved in polynomial multiplication (NTT, INTT)[4, Section 2.1]. This thesis will focus on optimizing the forward NTT and the inverse NTT for both round 1 and round 2 of Kyber. In this section we will ignore the $2n'th$ primitive root of unity ψ and the multiplication with n^{-1} from the definition of the NTT and INTT given in Section 2.2 as it is irrelevant at this point.

This paragraph is based on the Chapter 3 of the book *"Inside the FFT Black Box: Serial and Parallel Fast Fourier Transform Algorithms"* by E. Chu and A. George [14]. Efficient FFT and NTT implementations are based on the divide-and-conquer method and have a complexity of $O(n \log n)$, where n is the size of the input vector. The idea is to split the input vector of size n into two halves of size $n/2$, solve each part individually and then combine them into the result. By choosing $n = 2^k$, we can recursively apply this strategy $\log n$ times until we reach vectors of size one. Computing the NTT over individual halves requires the existence of the primitive $n/2$'th root of unity. For a vector of size n, we will explicitly refer to the primitive n't root of unity as ω_n. A property of the NTT when $n = 2^k$ is that the $n/2$'th root of unity exists and is $\omega_{\frac{n}{2}} = \omega_n^2$. Based on the way the vectors are split in two halves, we can differentiate two algorithms (known in the literature as butterflies):

- **The Cooley-Tukey (CT) butterfly [15]**. Also referred to as decimation in time, the CT algorithm splits the size-n input into one vector that contains the even-indexed values and another one that contains the odd-indexed values. After applying the NTT on both halves, let us denote by Y the vector obtained from the even indexes and with Z the vector obtained from the odd indexes. The full-size NTT vector X is obtained as follows: the first half of the resulting vector is computed as $X[i] = Y[i] + \omega_n^i Z[i]$ and the second half as $X[i + \frac{n}{2}] = Y[i] - \omega_n^i Z[i]$, with i from 0 to $n/2 - 1$. These two relations form the CT butterfly which is depicted in Figure 4.1 .

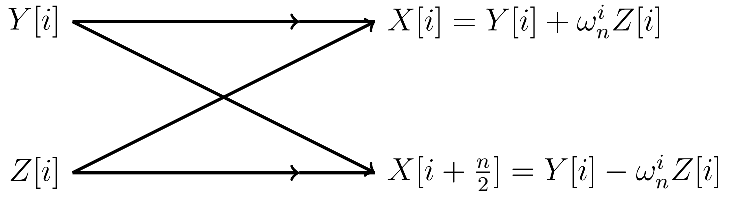
$Y[i]$
$Z[i]$
$X[i] = Y[i] + \omega_n^i Z[i]$
$X[i + \frac{n}{2}] = Y[i] - \omega_n^i Z[i]$

Figure 4.1: The Cooley-Tukey Butterfly.

- **The Gentleman-Sande (GS) butterfly [20]**. Also referred to as decimation in frequency, the GS algorithm splits the size-n input into one vector that contains the first $n/2$ values and another one that contains the last $n/2$ values. After applying the NTT on both halves, let us denote by Y the vector obtained from the first $n/2$ left indexes and with Z the vector obtained from the last $n/2$ right indexes. The final result X, is obtained as follows: the even indexes of the resulting vector are updated as $X[2i] = Y[i] + Z[i]$ and the second half as $X[2i + 1] = (Y[i] - Z[i])\omega_n^i$, with i from 0 to $n/2 - 1$. The CT butterfly is illustrated in Figure 4.2.

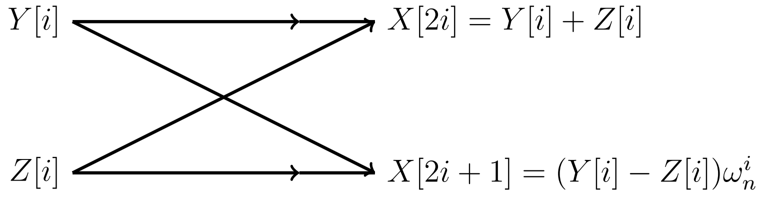

Figure 4.2: The Gentleman-Sande Butterfly.

In order to compute the NTT of a vector, the above-mentioned techniques are applied recursively until reaching vectors of degree one, for which $NTT(v) = v$. This process can be graphically illustrated in a binary tree of height $\log n$. The root of the graph represents the n-size input vector. Each parent of

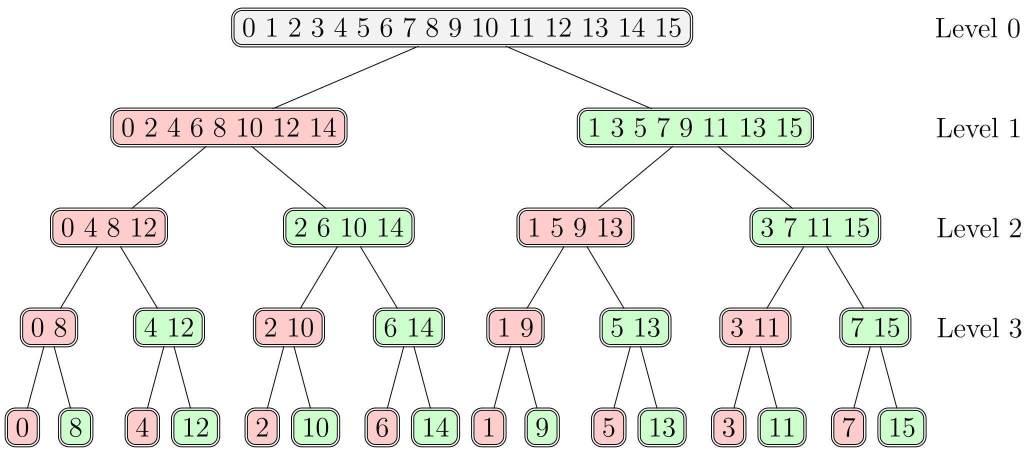

Figure 4.3: Cooley-Tukey NTT, $n = 16$

the graphs is obtained by combining its children according to either the CT butterfly or GS butterfly and using the powers of the right primitive root of unity corresponding to the size of the parent. The powers of the roots of unity are generally known in the literature as twiddle factors.

CT NTT example. For a better understanding on how to compute the NTT using the divide-and-conquer method, we will take a small example using a 16-coefficient polynomial as input. The binary tree in Figure 4.3 illustrates how a recursive NTT algorithm splits a vector of size 16 according to the Coley-Tukey butterfly. We say that all parent nodes (the leaves being excluded) that are situated at distance ℓ from the root, belong to level ℓ. Considering that the root is on level 0, the graph consists of $\log 16 = 4$ levels. Each level can be seen as rearranging the input values into smaller vectors according to the CT butterfly. Therefore, we will represent the values from the initial vector by their indexes so that we see how they propagate through the tree. The root of the tree contains the indexes of the initial vector (from 0 to 15). For each parent, the left child (red nodes) consists of the even-indexed values of the parent and the right child (green nodes) contains the odd-indexed values of the parent.

After splitting the input, we will start computing the NTT bottom-up according to the CT butterfly defined above ($X[i] = Y[i] + \omega_k^i Z[i]$, $X[i + \frac{k}{2}] = Y[i] - \omega_k^i Z[i]$), where X is the parent node of size k, Y is the left child and Z is the right child, both of size $k/2$. For the tree in discussion the leaves contain the indexes of the input vector, but in reality the butterflies will be applied on the actual values represented by these indexes. A parent of size k is computed with $k/2$ butterflies. However, the total number of butterflies in a level is always $n/2$, where n is the size of the initial input vector. In order to reconstruct a vector from level ℓ, we use $\omega_{\frac{n}{2^\ell}} = \omega_n^{2^\ell}$ as the primitive root of unity. Therefore, for reconstructing level 0, we use ω_{16}, for level 1 $\omega_8 = \omega_{16}^2$, for level 2 $\omega_4 = \omega_{16}^4$ and for level 3 $\omega_2 = \omega_{16}^8$.

Iterative NTT. In practice, these algorithms are transformed into their iterative version in order to eliminate the cost of $\log n$ recursive function calls. We take as example the same tree from Figure 4.3 so that we can derive rules to iteratively compute the $n/2$ butterflies for each level. Instead of processing multiple smaller vectors per level, we will now work with single size-n vectors, the leaves being represented by the initial vector. Let us denote the vector representing level ℓ with lev_ℓ.

In order to compute a level, the graph shows which indexes to combine from the previous level and at which indexes to store the results from each butterfly. For example, if we want to compute the values corresponding to the first node (0, 4, 8, 12) of level 2 using the children (0, 8) and (4, 12) we obtain: $lev_2[0] = lev_3[0] + lev_3[4]\omega_4^0$, $lev_2[8] = lev_3[0] - lev_3[4]\omega_4^1$ and $lev_2[4] = lev_3[8] + lev_3[12]\omega_4^0$, $lev_2[12] = lev_3[8] - lev_3[12]\omega_4^1$.

When computing a specific level ℓ, we can observe that the two input values of each butterfly are situated at the same distance of 2^ℓ from each other. For instance, the input values of the butterflies composing level 3 are (0, 8), (4, 12)...(7, 17), with a distance of $8 = 2^3$. Hence, we only need to know the values from the red nodes and we can obtain the corresponding green nodes values by adding 2^ℓ. Furthermore, we can group the butterflies by the twiddle factor they use. In other words, for ω^i we group all the values from the position i in the red nodes. If we order these values, we will obtain groups of consecutive numbers: for level 3 we get (0, 1, 2, 3, 4, 5, 6, 7), for level 2 (0, 1, 2, 3) and (8, 9, 10, 11), for level 1 (0, 1), (4 ,5), (8, 9) and (12, 13), and for level 0 (0), (2), (4), (6), (8), (10), (12) and (14). There are 2^ℓ consecutive numbers in a group and the distance between the first number in a group and the first from the next one is $2^{\ell+1}$.

The GS NTT can be represented in the same manner as the CT NTT, but in this case the red nodes will contain the first half values of the parent (the left indexes) and the green nodes will contain the second half values of the parent (the right indexes). This time we will count the levels bottom-up, such that the parents of the leaves will be on level 0 and the root on level $\log n - 1$. We will look again only at the red nodes values, as the green ones are obtained by adding 2^ℓ. Moreover, we can still group the butterflies by the twiddle factor they use. With the new level definition, the groups size is

$2^{\log n-\ell-1}$. The groups do not contain consecutive values this time, the gap between elements being $2^{\ell+1}$. The distance between the first number in a group and the first value from the next one is 1.

We will not explicitly describe the corresponding INTT algorithms as the only difference from the NTT algorithms is that the twiddle factors contain negative powers instead of positive ones.

As Kyber uses the negative wrapped convolution defined in Section 2.2, we have to take again into consideration the $2n'th$ primitive root of unity ψ and the multiplication with n^{-1} from the definition of Kyber's INTT. As a consequence, the only aspects that are changing in the above-described examples are the twiddle factors that now also take into account the second root of unity and the multiplication with the inverse of n at the end of the INTT.

4.2 Kyber Choices for the NTT

In order to save memory, Kyber computes the iterative (I)NTT's in-place, meaning that no auxiliary vectors are required to store intermediate results. As a consequence, both algorithms described above will produce the output in so-called "bit-reversed" order. For example, for a vector $\mathbf{a}$ = ($a_{0=00b}$, $a_{1=01b}$, $a_{2=10b}$, $a_{3=11b}$) with indexes on 2 bits, the bit-reversed order of **a** is **a'** = ($a_{0=00b}$, $a_{2=10b}$, $a_{1=01b}$, $a_{3=11b}$), where b indicates the binary representation of the index. This would typically imply the existence of an extra function that transforms a bit-reversed ordered vector into the normal ordered vector.

Another common optimization used in Kyber is to precompute and store the twiddle factors in memory. This is done in such a way that they can be fetched in the right order by the algorithms. Because of the negative wrapped convolution (defined in Section 2.2), the twiddle factors are a product between powers of ψ and powers of ω. Roy et al. [35] showed how to merge powers of ψ with powers of ω in a forward Cooley-Tukey NTT that accepts input in normal order and produces output in bit-reversed order. Similarly, Pöppelmann et al. [30] showed how to merge the powers of ψ^{-1} with the powers of ω^{-1} in an inverse Gentleman-Sande INTT that accepts input in bit-reversed order and produces the output in normal order. By using these two techniques, Kyber eliminates the need for additional transformations of the bit-reversed order into normal order. The following code listings are taken from the reference implementation of Kyber[1]. The CT forward NTT

[1]`https://github.com/pq-crystals/kyber`

used in Kyber round 1 is illustrated in Listing 4.2.1, the CT butterfly being described between the lines 11–17. The GS inverse NTT used in Kyber round 1 can be seen in Listing 4.2.2 with its specific butterfly between lines 11–19. Lines 24–25 of the GS algorithm reproduce the multiplication of the result with $n^{-1}\psi^{-i}$, the powers of ψ being fetched in bit-reversed order.

```
 1  void ntt(uint16_t *p){
 2  int level, start, j, k;
 3  uint16_t zeta, t;
 4
 5  k = 1;
 6  for(level = 7; level >= 0; level--){
 7   for(start = 0; start < KYBER_N; start = j + (1<<level)){
 8    zeta = zetas[k++];
 9    for(j = start; j < start + (1<<level); ++j){
10
11     t = montgomery_reduce((uint32_t)zeta * p[j + (1<<level)]);
12     p[j + (1<<level)] = barrett_reduce(p[j] + 4*q - t);
13
14     if(level & 1) // odd level
15      p[j] = p[j] + t; // Omit reduction (be lazy)
16     else
17      p[j] = barrett_reduce(p[j] + t);
18
19     }
20    }
21   }
22  }
```

Listing 4.2.1: CT NTT (Kyber round 1)

```c
void invntt(uint16_t * p){
int start, j, jTwiddle, level;
uint16_t temp, W;
uint32_t t;

for(level=0;level<8;level++){
 for(start = 0; start < (1<<level);start++){
  jTwiddle = 0;
  for(j=start;j<KYBER_N-1;j+=2*(1<<level)){
   W = omegas_inv_bitrev_montgomery[jTwiddle++];
   temp = p[j];

   if(level & 1) // odd level
    p[j] = barrett_reduce((temp + p[j + (1<<level)]));
   else
    p[j] = (temp + p[j + (1<<level)]); // Omit reduction (be lazy)

   t = (W * ((uint32_t)temp + 4*q - p[j + (1<<level)]));
   p[j + (1<<level)] = montgomery_reduce(t);
  }
 }
}

for(j = 0; j < KYBER_N; j++)
 p[j] = montgomery_reduce((p[j] * psis_inv_montgomery[j]));
}
```

Listing 4.2.2: GS INTT (Kyber round 1)

Because of the change in the definition of the NTT in round 2 (see Section 3.2), the code for the CT NTT and the GS INTT is also slightly different. As the vectors in Kyber are of size $n = 256$, the NTT's should have $8 = \log(256)$ levels which is the case for round 1. However in round 2 the new NTT definition requires only 7 levels. Aside from this adjustment, all the values in round 2 are represented as 16-bit signed integers instead of using 16-bit unsigned integers like in round 1. Round 2's butterflies are slightly modified as well. More precisely, they differ in the way Barrett and Montgomery reductions (both explained in the next sections) are applied. The C reference code of the CT NTT from Kyber round 2 is illustrated in Listing 4.2.3 with the CT butterfly between lines 13–15 and the C reference code of GS INTT can be seen in Listing 4.2.4 with the GS butterfly between lines 12–15. Also, at the end of the CT NTT a Barrett reduction is applied to each coefficient (see lines 17–18).

```
1  void ntt(int16_t r[256]) {
2  unsigned int len, start, j, k;
3  int16_t t, zeta;
4  
5  k = 1;
6  for(len = 128; len >= 2; len >>= 1) {
7   for(start = 0; start < 256; start = j + len) {
8    zeta = zetas[k++];
9    for(j = start; j < start + len; ++j) {
10    t = montgomery_reduce((int32_t)zeta*r[j + len]);
11    r[j + len] = r[j] - t;
12    r[j] = r[j] + t;
13   }
14  }
15 }
16 
17 for(j=0;j<KYBER_N;j++)
18  r[j] = barrett_reduce(r[j]);
19 
20 }
```

Listing 4.2.3: CT NTT (Kyber round 2)

```c
void invntt(int16_t r[256]) {
unsigned int start, len, j, k;
int16_t t, zeta;

k = 0;
for(len = 2; len <= 128; len <<= 1) {
 for(start = 0; start < 256; start = j + len) {
  zeta = zetas_inv[k++];
  for(j = start; j < start + len; ++j) {
   t = r[j];
   r[j] = barrett_reduce(t + r[j + len]);
   r[j + len] = t - r[j + len];
   r[j + len] = montgomery_reduce((int32_t)zeta*r[j + len]);
  }
 }
}

for(j = 0; j < 256; ++j)
 r[j] =montgomery_reduce((int32_t)r[j] * zetas_inv[127]);
}
```

Listing 4.2.4: GS INTT (Kyber round 2)

4.3 Modular Reductions

Because we are working with polynomials in R_q, all coefficients have to be reduced modulo q. As efficient modular reduction algorithms, Kyber uses Montgomery reduction [28] after multiplications and Barrett reduction [6] after additions and subtractions.

4.3.1 Montgomery Reduction

Montgomery reduction [28] is an algorithm used in fast multiplication modulo q of integers in Montgomery domain. We can transform an integer a into Montgomery domain (representation) by multiplying it with a factor R (to be defined later), followed by a reduction modulo q. When multiplying two numbers aR and bR in Montgomery domain we obtain $abR^2 \mod q$ which is not anymore in the domain. In order to convert the result back to the Montgomery representation we would multiply it with $R^{-1} \mod q$ and then we would still need to reduce it modulo q. By using the Montgomery reduction instead, we will get the same outcome, but more efficiently. To get numbers from Montgomery domain into the normal representation, we would apply again the Montgomery reduction.

For a modulus q and an integer $R = 2^k$, with $gcd(q, R) = 1$ and with $q' = -q^{-1} \mod R$, the Montgomery reduction of a consists of the following steps [23].

- Compute $u = a \cdot q' \mod R$,
- Compute $a \cdot R^{-1} \mod q$ as $(a + uq) \gg k$, where $\gg$ represents right shifting.

```
uint16_t montgomery_reduce(uint32_t a){
uint32_t u;

u = (a * qinv);
u &= ((1<<rlog)-1);
u *= q;
a = a + u;
return a >> rlog;
}
```

Listing 4.3.5: Unsigned Montgomery Reduction (Kyber round 1)

The trade-off in multiplying integers modulo q in Montgomery representation is that for each multiplication we first need to transform the numbers into the Montgomery domain, which is costly. As all multiplications inside the NTT involve a constant (a twiddle factor), Kyber can avoid this trade-off by precomputing and storing these values in Montgomery domain. When multiplied with a value outside the domain (a butterfly entry), the result will be of form abR, but not congruent modulo q. Because of the factor R, we can apply Montgomery reduction over the product and obtain the desired product $ab \mod q$.

```
int16_t montgomery_reduce(int32_t a) {
int32_t t;
int16_t u;

u = (int16_t)(a * qinv);
t = (int32_t)u * q;
t = a - t;
t >>= 16;
return (int16_t)t;
}
```

Listing 4.3.6: Signed Montgomery Reduction (Kyber round 2)

Round 1 of Kyber uses a variant of Montgomery reduction applied on unsigned numbers. This reduction is used in the Listing 4.2.1 at line 11 and in the Listing 4.2.2 at lines 19 and 25. For this round the prime q is 7681 and k is 18. Hence $R = 2^{18}$ and $q' = -7681^{-1} \mod 2^{18} \equiv 7679$. Kyber's C reference implementation of the unsigned Montgomery reduction can be viewed in Listing 4.3.5. In the listed code q' is identified as "qinv" and k as "rlog". For this round the input is needs to be a positive integer in the interval $\{0, ..., 2281446912\}$ and the output will be a positive integer in the interval $\{0, ..., 2^{13} - 1\}$.

Round 2 of Kyber uses a signed version of the algorithm listed in Listing 4.3.6. For this round the prime q is 3329, k is 16 and $R = 2^{16}$. For the signed algorithm we compute q' as $q' = q^{-1} \mod R$ so $q' = 3329^{-1} \mod 2^{18} \equiv 7679$. Again, in the code q' is identified as "qinv". For round 2 the input needs to be an integer in the interval $\{-2^{15}q, ..., 2^{15}q - 1\}$ and the output will be an integer in the interval $\{-q + 1, ..., q - 1\}$.

4.3.2 Barrett Reduction

Barrett reduction [6] is a well-known algorithm that efficiently computes $a \mod q$. An intuitive way of achieving this reduction would be to first divide a by q and then subtract the floor of the result from a. However, as divisions are expensive, Barrett's idea is to approximate $1/q$ by $m/2^k$ and replace the division with a right shift (i.e. $a/n = a \cdot m/2^k = (a \cdot m) \gg k$). The first step in computing Barrett reductions is to choose a k and set m to be the closest integer less or equal to $2^k/q$.

Round 1 of Kyber uses a variant of Barrett reduction applied on 16-bit unsigned integers. The C reference code of this algorithm is illustrated in

Listing 4.3.7. By choosing k=16, the algorithm is simplified to the formula $r = a - (a \gg 13) \cdot q$ and it outputs an integer modulo 7681 between 0 and a number on 14 bits, namely 11768 [12].

Round 2 of Kyber uses a variant of Barrett reduction applied on 16-bit signed integers. The corresponding C reference code can be seen in Listing 4.3.8. This time the value of k is 26, and the computation of m (variable v in the code) is approximated to the closes integer up instead of down. The function outputs an integer modulo 3329 between 0 and 3329.

```
uint16_t barrett_reduce(uint16_t a){
uint32_t u;

u = a >> 13;
u *= q;
a -= u;
return a;
}
```

Listing 4.3.7: Unsigned Barrett Reduction (Kyber round 1)

```
int16_t barrett_reduce(int16_t a) {
int32_t t;
const int32_t v = (1U << 26) / q + 1;

t = v * a;
t >>= 26;
t *= q;
return (int16_t)(a - t);
}
```

Listing 4.3.8: Signed Barrett Reduction (Kyber round 2)

Lazy Reduction. An additional optimization for round-1 Kyber is to apply Barrett reduction after the additions every second level instead of applying it every time. This is called lazy reduction. It is possible because the initial input is at most 14 bits and there is at most 1 bit of carry after each addition. So, after two addition the output can be of maximum 16 bits which fits in the 16-bit variables.

5. Kyber on RISC-V

5.1 RISC-V ISA

RISC-V is a relatively new open source instruction set architecture (ISA) having its origin at the University of California, Berkley. As the name suggests, it is based on RISC (Reduced Instruction Set Computer), meaning that it consists of a small number of simple instructions that can be put together to construct more complex behavior.

The RISC-V user-level (unprivileged) ISA includes a base integer instruction set that must be present in any implementation and some other optional instruction sets. At the moment of writing, the base integer ISA has two frozen versions: RV32I (for 32-bit machines) and RV64I (for 64-bit machines). Besides these, there are another two ISA's that have a draft status: RV128I for a 128-bit machine and RV32E a 32-bit version designed for micro-controllers. The standard extensions to the basic ISA are comprised of the integer multiplication and division ISA (denoted by M), the atomic ISA (denoted by A), the single-precision floating point ISA (denoted by F), the double-precision ISA (denoted by D) and the compressed instructions ISA. In this thesis we are using the RV32I ISA with the multiplication and division extension. The following lines will give an overview of the chosen instruction sets, based on the RISC-V instruction set manual, version 2.2 [38].

Registers. The RV32I ISA consists of 32 integer general purpose registers **x0–x31**, each having a fixed size of one word (32 bits). Depending on their purpose, these registers have specific names. For example, **x1** stores the return address **ra**, **x2** the stack pointer **sp**, **x4** the thread pointer **tp** and **x3** the global pointer **gp**. The register **x0** is also called **zero** because it always contains the value 0. Any write to this register is ignored. The function arguments are stored in the registers **x10–x17**, also known as **a0–a7**, and for the return values the registers **a0** and **a1** are reused. The temporary registers **t0–t6** correspond to **x5–x7** and **x28–x31**, and the rest of the registers **x8–x9**, **x18–x27** are used as programmer variables **s0–s11**.

Calling Conventions. One of the RISC-V's calling conventions is to pass the first eight integer and pointer arguments of a function into the registers **a0–a7** and if more arguments are needed, they are pushed on the stack. Similarly, the return value(s) of a function will be put in the registers **a0** and **a1**. The caller (the calling function) needs to save the values of the registers **a0–a7**, **t0–t7** and **ra** before calling a new function, otherwise their values will be lost. The calee (the function being called) can freely use the registers **a0–a7**, **t0–t7** and **ra**. If the registers **s0–s12**, **tp**, **sp** and **gp** are needed, the callee has to save their current values, and restore them at the end of its execution.

Instructions. RV32I has a reduced instruction set divided into several categories. The arithmetic and logical instructions are **add(i), sub, xor(i), or(i)** and **and(i)**. In general, the letter **i** from some instructions indicates an operation with an immediate. In RISC-V, the immediate can be represented on maximum 12 bits that are signed-extended to 32 bits before performing an operation. Because of this size restriction, there are two additional arithmetic instructions (**lui** and **auipc**) that help constructing up to 32-bit constants. To be noted that there is no subtraction with an immediate, as it can be translated into an addition with a negative immediate. The move instruction that is common in other assembly languages is replaced here with an addition with zero(i.e. mov rd,rs becomes add rd,rs,zero or addi rd,rs,0). Lastly, the bit-wise negation of a register r (i.e. not r) becomes xori r,-1.

Another type of instructions are loads and stores. The instructions belonging here are: **lb/sb** (load/store byte), **lh/sh** (load/store half-word) and **lw/sw** (load/store word). We can also choose to load an unsigned half-word (**lhu**) or an unsigned byte (**lbu**). Store and load instructions take as arguments one register keeping the value to be stored/loaded, another register containing the base address and an immediate representing the offset of the address. The only way of specifying the offset in a store/load instruction is through an immediate and it cannot be pre-computed and saved into a register.

The shift instructions comprises of logical left/right shifts (**sll(i),srl(i)**) and arithmetical shifts (**sra(i)**). RISC-V does not have any rotation instructions.

Branching is done through the conditional jump instructions **beq, bne, blt(u), bge(u)**. Each of these instructions compares the value of its first two operands (two registers) and if the condition is true it jumps to the address pointed by the third operand which is relative to the program counter. There are also two unconditional jump instructions: **jal** and **jalr**. The first one jumps to the destination address relative to the program counter and the second one to an absolute address given by a register.

The M extension covers multiplications, divisions and remainders. As multiplication of two 32-bit integers can result into a 64-bit integer, RISC-V requires two steps (instructions) to compute the complete result. One of them computes the lower 32 bits of the result (**mul**) and the other computes the upper 32-bits of the result. For a signed/unsigned multiplication the second used instruction is **mulhsu**, for a signed multiplication it is **mulh** and for an unsigned it is **mulhu**. Division is signed or unsigned (**div(u)**) and the signed/unsigned reminder can be obtained by **rem(u)**.

5.2 NTT Optimization on RISC-V

The focus of this thesis is to make the NTT and INTT faster by optimizing them at the assembly level on a RISC-V architecture. In other words, we aim to reduce the number of cycles these functions take. Common platform-independent practices for optimizing code for speed are to unroll loops, reduce memory access (loads and stores) and reduce the function calls and conditional statements. In addition, there are also platform-dependent optimization choices to be made based on the specific instruction set, the number of available registers, instruction latency, cache memory etc. In the RISC-V architecture there are 32 general-purpose registers from which only 29 are safe to rewrite (i.e., we cannot change the stack pointer as we will lose the position to the top of the stack). Compared to other architectures RISC-V has a significant number of usable registers (e.g. ARMv7E-M instruction set has only 14 usable registers [13, Section 2.3]). In consequence, we are able to both save constant values in registers such that we do not have to load them every time and to reduce memory access by loading multiple values at a time (see Appendix A for the registers map). More specific details will be given in the rest of this chapter. In general optimizing cryptography requires more attention than optimizing generic software as we need to be careful to not leak secret information via side channels. One of the most common side channel attacks that can be prevented from software are timing attacks. As Kyber does not use secret-dependent branches or lookup tables, this type of attacks are not feasible. As a consequence no special optimization techniques are required for the NTT and INTT. For a complete overview of Kyber and side channel attacks please have a look at [4, Section 4.5.2].

We start our optimization with the smallest blocks in the NTT/INTT (Barrett and Montgomery reductions) and then we build the butterflies on top of them. Finally, we optimize the full NTT/INTT. For a more readable code we use assembly macros to define the basic routines inside the NTT/INTT (for Montgomery reduction, Barrett reduction and the butterflies). Sometimes we need auxiliary registers to compute the block inside a macro. As it is not known which registers will be free when invoking a macro, we add the auxiliary registers to the arguments of the macro (the ones starting with \temp in the assembly listings) and decide later (at the "calling" time) which registers are free. In the reference C code, the 256-coefficient polynomials are represented by vectors of 256 values of 16 bits each, either unsigned (round 1) or signed (round 2). As the registers of RISC-V are on 32 bits and the values we are processing are on 16 bits, one could think it is efficient to load and store two values at a time into one register. However, the RV32I is designed to operate on 32-bit values and there are no instructions that facilitate processing the 16 upper/lower bits in parallel. Consequently, there is no gain in loading two values at a time as in the end we would need to process them one by one anyway. Besides, we would need to manually unpack them into two 32-bit registers, compute everything individually and repack them. The unpacking and packing introduce new instructions for extracting the lower and upper halves via bit-masking. Therefore, the most efficient way is to load and store 16 bits at a time which we do by using the instructions **lh, lhu** and **sh**.

5.2.1 Modular Reductions.

Listing 5.2.9 and Listing 5.2.10 illustrate our RISC-V assembly optimization for the unsigned version of the Barrett and Montgomery reductions in round 1, respectively for the signed version in round 2. For all presented macros the first argument represents the value to be reduced and it will also contain the result. The rest of the arguments are either the constants needed in these reductions or the auxiliary registers.

We would ideally pass all constants by value in the instructions that support an immediate operand. Considering that an immediate value in RISC-V is represented as 12 bits signed, the maximum positive value that a constant can take is 2047. Some constants from the Montgomery and Barrett reductions are bigger than this maximum value and thus they cannot be used as operands. Because RISC-V has a sufficient number of usable registers we permanently store the constants bigger than 2047 in registers during the entire execution of one NTT/INTT. In this way we avoid loading them every time we call a reduction macro.

All the multiplications inside the reductions are dealing with 16-bit numbers, therefore we would only need the **mul** instruction to compute the lowest 32 bits. If the result has to be only 16 bits long, we need to truncate it and for the signed representation to also sign-extend it. In general our assembly implementation for the reductions follows intuitively from the corresponding C reference code (see section 4.2). In the reference code for round 1 all 16-bit unsigned multiplications inside the reductions result into a 32-bit integer, so no further operations are needed.

For the Barrett reduction in round 2, both multiplications are stored into a 32-bit signed variable and by only using the **mul** instruction we already obtain the correct result with no further processing. The right shift from the reference implementation is translated into an arithmetic right shift in the assembly (i.e. it maintains the sign of the number). This is valid for all right shifts that we are performing on signed integers.

According to the reference code of round 2, the result of the first multiplication in the Montgomery reduction must be a 16-bit signed number. What we would typically do is to use the **mul** instruction to get the first 32 bits of the result, discard the most significant 16 bits, then bring the result back to the original position and sign extend it to 32-bits. We would do this by shifting the result 16 bits to the left and then using a 16-bit arithmetic right shift. In the end this multiplication would take 3 instructions. However, we can optimize it such that we eliminate the shift operations. The first step is to shift the values of q and qinv with 16 bits to the left and store these values in registers so that we only compute these operations once (for the registers map please have a look in Appendix A). We will name these values $q\_2 = q \ll 16$ and $qinv\_2 = qinv \ll 16$. In this way the lowest 16 bits of these constants will be 0 and the highest 16 bits will contain the actual value. The first multiplication will now compute $a \cdot qinv\_2$ which gives the same result as before, but positioned in the upper half of the result register instead of in the lower half. The advantage is that the result is now automatically truncated to 16 bits and the sign bit of the result coincide with the sign bit of the register, so no sign-extension needed. The next step is to multiply this outcome with the value of q_2. Because both 32-bit operands have the least significant 16 bits equal to 0, the first 32 bits of the result will also be 0. In conclusion we are interested in the highest 32 bits of the result which we obtain by using the instruction **mulh**. No further steps are required as in the reference code this result is a signed integer on 32 bits, the same as the output of the **mulh** instruction.

```
.macro montgomery a,qinv,q,
temp0,temp1,mont_const

mul \temp0,\a,\qinv
and \temp0,\temp0,\mont_const
mul \temp1,\temp0,\q
add \a,\a,\temp1
srli \a,\a,18
.endm



.macro barrett a,q,
temp0,temp1

srli \temp0,\a,13
mul \temp1,\temp0,\q
sub \a,\a,\temp1
.endm
```

Listing 5.2.9: Unsigned Montgomery and Barrett (round 1)

```
.macro montgomery_2 a,qinv_2,
q_2,temp0,temp1

#qinv_2 is qinv<<16
#q_2 is q<<16
mul \temp0,\a,\qinv_2
mulh \temp1,\temp0,\q_2
sub \a,\a,\temp1
srai \a,\a,16
.endm

.macro barrett_2 a,q,
const,temp0,temp1

mul \temp0,\const,\a
srai \temp0,\temp0,26
mul \temp1,\temp0,\q
sub \a,\a,\temp1
.endm
```

Listing 5.2.10: Signed Montgomery and Barrett (round 2)

5.2.2 Butterflies.

Listing 5.2.11 and Listing 5.2.12 list our RISC-V approach for the butterflies in round 1 and round 2, respectively. The first two arguments of the butterflies represent the two input values to be processed and the third one is the corresponding twiddle factor. Because we are calling the Barrett and Montgomery reductions inside the butterflies, we also need the constants these macros use and auxiliary registers. Therefore the rest of the butterfly arguments lie in this category.

The fist macro of each listing refers to the CT butterfly and the second macro of each listing refers to the GS butterfly (we call it in the code butterfly_inv as it belongs to the INTT). For round 1 we remove the conditional statement used for the lazy reduction in both CT and GS butterflies by splitting each of them into one that is applied in odd levels (odd butterfly) and one that is applied in even levels (even butterfly). For the CT butterfly we apply Barrett reduction after computing the right term when the level is even, whereas for the GS butterfly we apply a Barrett reduction after computing the left term when the level is odd. For round 2 there is no lazy reduction so there is only one version of butterfly for the forward NTT and one version for the INTT.

```
.macro butterfly_even left,right,zeta,q4,qinv,q,temp0,temp1,
t,mont_const

mul \t,\zeta,\right
montgomery \t,\qinv,\q,\temp0,\temp1,\mont_const
add \right,\left,\q4
sub \right,\right,\t
add \left,\left,\t
barrett \right,\q,\temp0,\temp1
  #Even level only
  barrett \left,\q,\temp0,\temp1
.endm



.macro butterfly_inv_odd left,right,omega,q4,qinv,q,temp0,
temp1,temp2,mont_const

add \temp2,\left,x0
add \left,\left,\right
  #Odd level only!
  barrett \left,\q,\temp0,\temp1
add \temp2,\temp2,\q4
sub \temp2,\temp2,\right
mul \right,\omega,\temp2
montgomery \right,\qinv,\q,\temp0,\temp1,\mont_const
.endm
```

Listing 5.2.11: Butterflies Round 1

```
.macro butterfly2 left,right,zeta,qinv_2,q_2,temp0,temp1,temp2
mul \temp0,\zeta,\right
montgomery_2 \temp0,\qinv_2,\q_2,\temp2,\temp1
sub \right,\left,\temp0
add \left,\left,\temp0
.endm



.macro butterfly2_inv left,right,zeta,qinv_2,q,bar_const,temp0,
temp1,temp2,q_2

add \temp0,\left,x0
add \left,\left,\right
barrett_2 \left,\q,\bar_const,\temp1,\temp2
sub \temp0,\temp0,\right
mul \right,\temp0,\zeta
montgomery_2 \right,\qinv_2,\q_2,\temp0,\temp1
.endm
```

Listing 5.2.12: Butterflies Round 2

5.2.3 NTT and INTT.

Once we have optimized the reductions and the butterflies, we integrate them into the NTT/INTT. The first step we take is to unroll the outer layer of the NTT/INTT. More specifically we obtain one NTT/INTT loop for each level (8 levels for round 1 and 7 levels for round 2). For round 1 we use the odd butterflies in the odd levels and the even butterflies in the even levels. At this point we aim to reduce the number of load and store instructions. We know that for each level, after we select two matching entries to compute a butterfly, we put the result back in the same spots. What we can do instead of storing the result back is to look at the next consecutive level at the same two positions and see for each of them which is the other corresponding butterfly component. We can see that these two new positions form a butterfly in the previous level. For example, if we want to merge level 7 and 6 for our 256-coefficient NTT we can see that position 0 matches position 128 in level 7. In level 6, 0 matches 64 and 128 matches 192, but 64 and 192 are also a pair in level 7. Therefore, we could merge two levels by iterating 64 times (256/4) over the input and loading 4 values at a time. We would compute 4 butterflies (2 per level) and afterwards store the 4 results.

Following this logic and given the large number of registers in RISC-V, we are able to merge a maximum of 4 consecutive levels and load 16 values at a time, computing 32 butterflies (8 per level) before storing the 16 results. For the NTT we merge levels 7–4 and 3–0 for round 1 and 7–4 and 3–1 for round 2. Similarly, for the INTT we merge levels 0–3 and 4–7 for round 1 and levels 1–3 and 4–7 for round 2. Table 5.2.3 illustrates how we are choosing the first 16 values to be loaded when merging the NTT levels 7 (top row of the table) to 4 (bottom row of the table). Each sequence of red numbers in the table is matched with the succeeding sequence of blue numbers. We can see that the distance between two consecutive loaded values is 16. As the load and store instructions use a base address and an offset we only need to know the address of the first value to be loaded and use the values in the table as offsets to load all 16 butterflies values. In order to cover all 256 values of the NTT we iterate over the first 16 elements of the input vector which we use as base addresses.

0	16	32	48	64	80	96	112	128	144	160	176	192	208	224	240
0	16	32	48	64	80	96	112	128	144	160	176	192	208	224	240
0	16	32	48	64	80	96	112	128	144	160	176	192	208	224	240
0	16	32	48	64	80	96	112	128	144	160	176	192	208	224	240

Table 5.1: Index matching for merging NTT levels 7–4

After each level, the distance between two butterfly entries halves, therefore from level 3 to 0 these distances are 8, 4, 2 and 1. This means that in order to merge levels 3–0 (or 3–1) we will first load the values from the positions 0 to 15. We iterate through the input vector with the step 16 to get the base addresses and we load the butterflies values using the offsets 0 to 15. For round 2 we do not compute level 0, so we are computing only 24 butterflies between loading and storing the 16 values.

For the INTT we merge the levels in the same way as for the NTT. The difference is that now the levels start from 0 to 7 (or 1 to 7), meaning that we first compute levels 0–3 (or 1–3) and then levels 4–7. Table 5.2.3 shows how we choose the first 16 indices for merging levels 1–3 in round 2. For the NTT we do the same, but instead we interpret the same table bottom-up.

After computing all levels in the INTT in round 1 and round 2, a multiplication by $n^{-1}\psi^{-i}$ (see the definition of INTT in section 2.2 and the reference

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Table 5.2: Index matching for merging INTT round 2, levels 1–3

code in section 4.2) followed by a Montgomery reduction is performed for each coefficient of the resulting vector. This implies that 256 values are loaded, processed and then stored back. We eliminate these additional loads and stores by integrating these operations into the merged block for levels 4–7. More specifically, we compute the multiplication and then the Montgomery reduction after computing level 7 (the last one) of the INTT and before storing the results. Similarly, we integrate the Barrett reductions that are applied at the end of the NTT round 2 for each coefficient. We integrate it in the merged block for levels 3–1 after computing the butterflies of level 1. We again eliminate 256 load and 256 store instructions.

What is left to discuss is how we integrate the twiddle factors. We use the same pre-computed twiddle vectors as in the reference code. Because we do not have registers left to store these values beforehand, we load them each time we need them. Thus for levels 3 and 7 we load 1 twiddle and compute 8 butterflies with it, for levels 2 and 6 we load 2 twiddles and computer 4 butterflies with each. In levels 1 and 5 we load 4 twiddles and compute 2 butterflies with each and in levels 0 and 4 we load 8 twiddles and compute 1 butterfly with each. For the NTT in both rounds and for the INTT in round 2, levels 4–7 require in total 15 different twiddle factors. The same levels in round 1 INTT use in total 8 different twiddle factors: 1 of them in level 7, 2 in level 6, 4 in level 5 and all of them in level 4. For the levels 4–7 these values are loaded in consecutive order from the input vector. This is not the case for levels 0(1) to 3 which makes it difficult to load the correct twiddles as we do not have enough free registers to compute the right position. As a solution, we rearrange the twiddle factors in a new vector for each NTT/INTT such that we can fetch them one after another without any computations in the assembly code. We use a Python script to generate the new twiddle factors from the input vector. In total, levels 0–3 use 240 twiddles and levels 1–3 use 128 twiddles, thus these are also the sizes of the generated vectors.

5.2.4 Instruction Interleaving Round 2

In the simulator where we test out code (see Chapter 6) we observed that not all instructions have the same latency. After several tests we concluded that most of the instructions that we are using have a latency of 1 cycle, with some exceptions: the instructions **sra(i), sll(i)** have a latency of 2 cycles and the instructions **mul, mulh, lh(u), lw, sh, sw** have a latency of 3 cycles. As a reminder, latency is the time (cycles) we need to wait for an instruction such that the result is ready to use. If we do not need the result earlier than their latency, these instructions will count as 1 cycle.

As during the work on this thesis Kyber has been updated to round 2, we are focusing more on the latest version for which we also take into account the instructions latency. The optimization technique that we use in this case is to interleave instructions such that while we wait for a result we can use other instructions that do not depend on that result. For example if we use a 3-cycle latency instruction followed by a result-dependent 1-cycle instruction, the total cycle count will be 4 (3+1). If instead we add 2 1-cycle result-independent instructions after, the total cycle count will be 3 (1+1+1).

Our approach for the NTT and INTT in round 2 is to interleave multiple butterflies, Barrett reductions and Montgomery reductions. In principle if we want to interleave more macros, we take the first instruction from each and put them one after another, then we do the same for the rest of the instructions. Because the largest latency we encounter is 3, we obtain the highest optimization level by interleaving at least 3 macros. In this way we reduce the cycle count of the highest latency instruction from 3 to 1. A downside in merging multiple macros is that it requires more auxiliary registers. Even though we are already using most of the registers, we can still find sufficient free registers at the time of calling our merged macros.

For both NTT and INTT, for each level, we merge the butterflies that use the same twiddle factor. We can do this by using few extra auxiliary registers as we have all the values we need already stored in registers: the butterflies values, the twiddle factor and the reductions constants. When grouping the butterflies by the twiddle factor we can merge 2, 4 or 8 butterflies.

In the NTT we do not have any Barrett reductions inside the butterflies, but we compute 16 Barrett reductions at the end of level 0. We manage to reduce the cycle count of all instructions in the Barrett reductions to 1, by interleaving 4 of them and applying the merged macros 4 times. We cannot interleave the Montgomery reductions because we do not have registers left for this purpose.

In the INTT however, both Montgomery and Barrett reductions are performed at the end of computing the left and the right result of the butterflies. Therefore we can remove the reductions from the butterflies and apply all the reductions at the end of the level. As there are 16 values in total and half of them need a Barrett reduction and the others a Montgomery reduction, we are merging 4 Barrett reductions and 4 Montgomery reductions and we apply each of them 2 times. By doing this we again reduce all instructions cycle count from the reductions to 1. For the multiplication followed by a Montgomery reduction at the end of level 7, we first compute all multiplications, and then all reductions. If we compute the multiplications one after another we do not use their result immediately, therefore we reduce all multiplications from 3 cycles to 1. For the Montgomery reductions we call 4 times 4 interleaved Montgomery macros. A single Montgomery reductions takes 8 cycles when the result is not immediately used, and 9 cycles if the result is immediately used. Thus 4 independent Montgomery reductions use 32/36 cycles, while 4 interleaved Montgomery reductions use 16 cycles. Similarly 4 Barrett independent reductions will take 36 ($4 \cdot 9$) cycles and 4 interleaved Barrett reductions will take 16 cycles.

Moreover, when we load and store the 16 butterflies values together we automatically use the interleaving technique. This is because we are applying these 16 operations one after another and in this way we do not use their result immediately. Thus, their cycle count will be 1 instead of 3.

6. Results

Testing Environment. Before presenting our results, we mention that we integrate our code into the *pqriscv*[1] framework and we use it as a starting point for compilation and testing. This is a work-in-progress repository which currently aims to integrate and benchmark multiple post-quantum schemes implementations on RISC-V. We start our optimization from the reference code of Kyber[2] and we use the Keccak included with *pqriscv*. Although the Keccak permutation has recently been optimized for the RISC-V architecture in [37] and it is faster than the permutation we currently use, the overall speed of the optimized Keccak is slower compared to the version we currently use. This is because the optimized permutation requires changing the rest of the Keccak implementation to perform bit interleaving which results into more computations.

We run our code on a 32-bit RISC-V CPU simulator taken from *PQVexRiscV*[3]. This repository is a test-platform created specifically for the *pqriscv* project and it is based on *VexRiscv* [4]. The latter is a public 32-bit RISC-V CPU implementation written in *SpinalHD*L[5].

Speed. We compile our code using the `riscv64-unknown-elf-gcc` command from the RISC-V GNU toolchain[6] with the following compiling options: `-O3 -fno-common`. These options are taken from the Makefile of the *pqriscv* project. For measuring the speed we use the `get_cycles()` function before

[1]`https://github.com/mupq/pqriscv`

[2]`https://github.com/pq-crystals/kyber/tree/master/ref`

[3]`https://github.com/mupq/pqriscv-vexriscv`

[4]`https://github.com/SpinalHDL/VexRiscv`

[5]`https://spinalhdl.github.io/SpinalDoc-RTD/`

[6]`https://github.com/riscv/riscv-gnu-toolchain`

and after the section of code we want to measure. We then compute the difference between the second and the first measurement and we obtain the cycle count. The speed results for our optimized assembly NTT and INTT for round 1 and 2 can be seen in Table 6.1. As there is no other implementation of the NTT and INTT of Kyber on RISC-V (as far as we know) to compare our results with, we also include in this table the cycle count of the corresponding reference code[7]. Although the reference code is not meant to be fast, we want to know where to place our results (e.g., if our code turns out to be slower than the reference implementation, then we clearly do something wrong). We can see in the table that our optimized version of the NTT and INTT for both round 1 and 2 is (70–83 %) faster than the corresponding reference code, which is a significant improvement.

The performance of Kyber with our optimized assembly integrated can be seen in Table 6.2 and Table 6.3. These tables show the cycle count for key generation, key encapsulation and key decapsulation for round 1 and 2. As it can be seen in the table, Keccak takes between 63–76 percent of the total cycle count of each block, which is more than half of the scheme.

Round	**NTT optim**	**INTT optim**	**NTT ref**	**INTT ref**
1	27 390 (-71.7%)	25 669 (-70%)	96 719	82 632
2	14 348 (-72.3%)	13 742 (-82.4%)	51 832	78 067

Table 6.1: NTT and INTT cycle count for round 1 and 2

[7]`https://github.com/pq-crystals/kyber`

Implementation	**Measurement**	**Key Gen**	**Encaps**	**Decaps**
Kyber512	**Total cycles**	1 504 433	2 005 396	1 954 806
	Keccak cycles	1 130 444	1 486 761	1 261 821
	Keccak %	75.14	71.14	64.55
Kyber768	**Total cycles**	2 494 852	3 091 996	3 005 500
	Keccak cycles	1 886 168	2 305 120	1 986 925
	Keccak %	75.6	74.55	66.11
Kyber1024	**Total cycles**	3 914 528	4 645 965	4 560 553
	Keccak cycles	2 993 751	3 505 874	3 125 235
	Keccak %	76.48	75.46	68.53

Table 6.2: Kyber round 1 cycle count

Implementation	Measurement	Key Gen	Encaps	Decaps
Kyber512	**Total cycles**	1 218 557	1 592 689	1 515 876
	Keccak cycles	885 012	1 180 015	954 931
	Keccak %	72.63	74.09	63.0
Kyber768	**Total cycles**	2 288 109	2 771 517	2 653 584
	Keccak cycles	1 701 782	2 090 040	1 771 623
	Keccak %	74.38	75.41	66.76
Kyber1024	**Total cycles**	3 686 344	4 280 420	4 123 722
	Keccak cycles	2 779 870	3 261 683	2 849 933
	Keccak %	75.41	76.2	69.11

Table 6.3: Kyber round 2 cycle count

Stack Usage. We report the stack size for Kyber in bytes in Table 6.4. To measure the stack size we use the stack.c file from the *mupq*[8] project. The stack is measured by filling it with canaries (a fixed value) before a function call and after counting how many bytes are overwritten. The stack size of the NTT of round 1 is 60 bytes (we push 13 registers in the beginning of the function $\Longrightarrow 13 \cdot 4B = 52B$). The stack size of the round 2 NTT and round 1 and 2 INTT is 64 bytes (we push 14 registers in the beginning of the function $\Longrightarrow 14 \cdot 4B = 56B$). There are still 8 bytes from the stack for each function that remain unexplained, which are the return address (4 bytes) and the base pointer (another 4 bytes).

Implementation	**Round 1**			**Round 2**		
	KeyGen	**Encaps**	**Decaps**	**KeyGen**	**Encaps**	**Decaps**
Kyber512	6 648	9 304	10 120	6 632	9 288	10 072
Kyber768	10 744	13 912	15 080	10 728	13 896	15 032
Kyber1024	15 864	19 544	21 064	15 848	19 528	21 144

Table 6.4: Kyber stack size (in bytes) round 1 and 2

Code Size. The code size of the NTT, INTT and the whole Kyber scheme are shown in Table 6.5. In order to get the total code size of Kyber, we compile individually all files in Kyber with the `-o` option in order to generate object files. Then, we add all object files into an archive with the command `riscv64-unknown-elf-ar -crs`. We measure the overall size by using the command `riscv64-unknown-elf-size -t` on the previously-created archive. The code size of Keccak is 7678 bytes (this number includes the files fips202.c and keccakf1600.c from *pqriscv*).

Round	**NTT**	**INTT**	**Full Kyber**
1	3 680	3 468	62 718
2	2 160	3 328	59 974

Table 6.5: Kyber code size (in bytes) round 1 and 2

Comparison. As far as we know there is no other software optimization of the NTT and INTT of Kyber on a RISC-V architecture. This makes it impossible to compare our results with others obtained on the same architecture. Regarding other architectures, simply taking our results and comparing

[8]`https://github.com/mupq/mupq/tree/e41de1a97e51394ee15db80380b2659bb109b552/crypto_kem`

them with the cycle counts from a different architecture is meaningless. For example, completing 1 cycle can take different amount of time for different architectures (frequency of the clock). Also, two architectures with different number of registers, different instruction latency and even different instructions in general can result into two distinct optimization techniques for the same functionality. All these make it hard to compare the performance of the same functionality on different architectures.

However, if the reader is interested in other speed optimization of Kyber on different architectures, one example is [13, Section 2.3], which targets the Arm Cortex-M4 architecture. For round 1 their Arm Cortex-M4 optimized NTT is 9452 cycles and the INTT is 10373 cycles. For round 2 their NTT takes 7725 cycles and the INTT 9347 cycles. What is common for the implementations of both architectures (Arm Cortex-M4 and RISC-V) is that Keccak takes more than half of the total cycles of Kyber (even with an optimized version for Arm Cortex-M4). As a general remark, we can also observe that compared to both Arm Cortex-M4 implementation and to the reference code, in our optimization of round 2 the INTT is faster than the NTT. This is due to our interleaved implementation in which we remove the Montgomery and Barrett reductions from the butterflies and merge multiple of them only after computing 8 butterflies (see Section 5.2.4). For the stack size, we can see that ours is way bigger than the one in the Arm Cortex-M4 implementation, but we do not optimize anything there, while they do decrease the stack usage [13, Section 4].

7. Conclusion

In this thesis we have optimized round 1 and round 2 of Kyber for speed on the RISC-V architecture. We have focused on the NTT and INTT together with the Montgomery and Barrett reductions inside these functions. As RISC-V has a large number of reusable registers (29), our optimization approach was to merge up to 4 levels of the NTT and INTT and significantly reduce the store and load instructions and implicitly the overall cycle count. In this way, we could load 16 polynomial coefficients at a time and compute 8 butterflies per level, meaning that for 4 merged levels we computed 32 butterflies before storing back the results, and for 3 merged levels, we computed 24 butterflies before storing the values back. With this optimization, for round 1, instead of using 2048 (8 levels $\cdot$ 256 coefficients) load and 2048 store instructions, we only used 512 ($2 \cdot 256$) of each. In the same way, for round 2, instead of using 1792 (7 levels $\cdot$ 256 coefficients) load and 1792 store instructions, we only used 512 ($2 \cdot 256$) of each. Moreover, for the INTT of both rounds and for the NTT of round 2 we saved another 256 store and 256 load instructions. We did this by computing the multiplication by $n^{-1}\psi^{-i}$ at the end of the INTTs and the Barrett reductions at the end of the NTT within the last merged block of the INTT, respectively NTT. In addition, we optimized the Barrett and Montgomery reductions for round 1 and 2 and integrated them inside the NTT and INTT. Lastly, for round 2 we took into account the instructions latency, and we significantly reduced the total cycle count by interleaving multiple of them.

With the presented techniques, we obtained for round 1 an NTT of 27390 cycles and an INTT of 25669 cycles. For round 2 we obtained an NTT of 14348 cycles and an INTT of 13742 cycles. As there is no other software implementation of the NTT and INTT of Kyber on RISC-V, we have compared our speed optimization results with the results from the reference implementation, even though the latter is not optimized in any ways. However, our NTT and INTT cycle count is around 70 – 80 % faster for both rounds of Kyber, which is at least way better than the reference implementation.

As in this thesis we have put more effort in the round 2 of Kyber, it would be interesting to see in a future work how much we can still optimize round 1 by using the interleaving technique. Another aspect that is worth improving is the performance of Keccak. We have seen that Keccak takes more than 60% of each Kyber block (key generation, encapsulation and decapsulation) for both rounds. A direction in improving Keccak would be to fully optimize it in RISC-V assembly and see if we can make it faster.

References

[1] E. Alkim, L. Ducas, T. Pöppelmann, and P. Schwabe. Post-quantum key exchange—a new hope. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 327–343, Austin, TX, Aug. 2016. USENIX Association.

[2] B. Applebaum, D. Cash, C. Peikert, and A. Sahai. Fast Cryptographic Primitives and Circular-Secure Encryption Based on Hard Learning Problems. In S. Halevi, editor, *Advances in Cryptology - CRYPTO 2009*, pages 595–618, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[3] F. Arute, K. Arya, R. Babbush, D. Bacon, J. C. Bardin, R. Barends, R. Biswas, S. Boixo, F. G. Brandao, D. A. Buell, et al. Quantum supremacy using a programmable superconducting processor. *Nature*, 574(7779):505–510, 2019.

[4] R. Avanzi, J. Bos, L. Ducas, E. Kiltz, T. Lepoint, V. Lyubashevsky, J. M. Schanck, P. Schwabe, G. Seiler, and D. Stehlé. CRYSTALS-Kyber Algorithm Specifications And Supporting Documentation. *Submission to the NIST post-quantum project*, 9:11, 2017. `https://pq-crystals.org/kyber`.

[5] R. Avanzi, J. Bos, L. Ducas, E. Kiltz, T. Lepoint, V. Lyubashevsky, J. M. Schanck, P. Schwabe, G. Seiler, and D. Stehlé. CRYSTALS-Kyber Algorithm Specifications And Supporting Documentation (version 2.0). *Submission to round 2 of the NIST post-quantum project*, 2019. `https://pq-crystals.org/kyber`.

[6] P. Barrett. Implementing the Rivest Shamir and Adleman Public Key Encryption Algorithm on a Standard Digital Signal Processor. In A. M.

Odlyzko, editor, *Advances in Cryptology — CRYPTO' 86*, pages 311–323, Berlin, Heidelberg, 1987. Springer Berlin Heidelberg.

[7] A. Bauer, H. Gilbert, G. Renault, and M. Rossi. Assessment of the key-reuse resilience of newhope. Cryptology ePrint Archive, Report 2019/075, 2019. `https://eprint.iacr.org/2019/075`.

[8] M. Bellare, A. Desai, D. Pointcheval, and P. Rogaway. Relations among notions of security for public-key encryption schemes. In H. Krawczyk, editor, *Advances in Cryptology — CRYPTO '98*, pages 26–45, Berlin, Heidelberg, 1998. Springer Berlin Heidelberg.

[9] S. Benz. *Fast multiplication of multiple-precision integers.* PhD dissertation, Rochester Institute of Technology, 1991. `https://pdfs.semanticscholar.org/6c84/f11da1d7831fd78226892078b5d1c7304fd4.pdf`.

[10] D. J. Bernstein, L. G. Bruinderink, T. Lange, and L. Panny. HILA5 pindakaas: On the CCA security of lattice-based encryption with error correction. Cryptology ePrint Archive, Report 2017/1214, 2017. `https://eprint.iacr.org/2017/1214`.

[11] G. Bertoni, J. Daemen, M. Peeters, and G. Van Assche. The Keccak reference. *Submission to the NIST SHA-3 competition*, 2011. `https://keccak.team/files/Keccak-reference-3.0.pdf`.

[12] J. Bos, L. Ducas, E. Kiltz, T. Lepoint, V. Lyubashevsky, J. M. Schanck, P. Schwabe, G. Seiler, and D. Stehlé. CRYSTALS-Kyber: a CCA-secure module-lattice-based KEM. In *2018 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 353–367. IEEE, 2018.

[13] L. Botros, M. J. Kannwischer, and P. Schwabe. Memory-Efficient High-Speed Implementation of Kyber on Cortex-M4. In J. Buchmann, A. Nitaj, and T. Rachidi, editors, *Progress in Cryptology – AFRICACRYPT 2019*, pages 209–228, Cham, 2019. Springer International Publishing.

[14] E. Chu and A. George. *Inside the FFT BLACK BOX: Serial and Parallel Fast Fourier Transform Algorithms.* CRC Press, 2000.

[15] J. W. Cooley and J. W. Tukey. An algorithm for the machine calculation of complex Fourier series. *Mathematics of computation*, 19(90):297–301, 1965.

[16] W. Diffie and M. Hellman. New directions in cryptography. *IEEE transactions on Information Theory*, 22(6):644–654, 1976.

[17] M. J. Dworkin. Permutation-based Hash and Extendable-Output Functions, 2105. http://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.202.pdf.

[18] P. FIPS. 197: Advanced encryption standard (AES). *National Institute of Standards and Technology*, 26, 2001. https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.197.pdf.

[19] E. Fujisaki and T. Okamoto. Secure Integration of Asymmetric and Symmetric Encryption Schemes. In M. Wiener, editor, *Advances in Cryptology — CRYPTO' 99*, pages 537–554, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg.

[20] W. M. Gentleman and G. Sande. Fast Fourier Transforms: for fun and profit. In *Proceedings of the November 7-10, 1966, fall joint computer conference*, pages 563–578. ACM, 1966.

[21] J. Hoffstein, J. Pipher, and J. H. Silverman. NTRU: A ring-based public key cryptosystem. In J. P. Buhler, editor, *Algorithmic Number Theory*, pages 267–288, Berlin, Heidelberg, 1998. Springer Berlin Heidelberg.

[22] S. R. Jintai Ding, Scott Fluhrer. Complete Attack on RLWE Key Exchange with reused keys, without Signal Leakage. Cryptology ePrint Archive, Report 2017/1185, 2017. https://eprint.iacr.org/2017/1185.

[23] J. Katz, A. J. Menezes, P. C. Van Oorschot, and S. A. Vanstone. *Handbook of applied cryptography.* CRC press, 1996.

[24] A. Langlois and D. Stehlé. Worst-case to average-case reductions for module lattices. *Designs, Codes and Cryptography*, 75(3):565–599, 2015.

[25] V. Lyubashevsky, C. Peikert, and O. Regev. On Ideal Lattices and Learning with Errors over Rings. In H. Gilbert, editor, *Advances in Cryptology – EUROCRYPT 2010*, pages 1–23, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[26] V. Lyubashevsky and G. Seiler. NTTRU: Truly Fast NTRU Using NTT. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2019(3):180–201, 2019.

[27] R. J. McEliece. A Public-Key Cryptosystem based on Algebraic Coding Theory. *DSN Progress Report*, pages 114–116, 1978.

[28] P. L. Montgomery. Modular multiplication without trial division. *Mathematics of computation*, 44(170):519–521, 1985.

`http://www.ams.org/journals/mcom/1985-44-170/S0025-5718-1985-0777282-X/S0025-5718-1985-0777282-X.pdf`.

[29] M. Nielsen and I. Chuang. Quantum computation and quantum information, 2002.

[30] T. Pöppelmann, T. Oder, and T. Güneysu. High-Performance Ideal Lattice-Based Cryptography on 8-Bit ATxmega Microcontrollers. In K. Lauter and F. Rodríguez-Henríquez, editors, *Progress in Cryptology – LATINCRYPT 2015*, pages 346–365. Springer International Publishing, 2015.

[31] Y. Qin, C. Cheng, and J. Ding. A complete and optimized key mismatch attack on nist candidate newhope. Cryptology ePrint Archive, Report 2019/435, 2019. `https://eprint.iacr.org/2019/435`.

[32] O. Regev. On lattices, learning with errors, random linear codes, and cryptography. *Journal of the ACM (JACM)*, 56(6):34, 2009.

[33] O. Regev. The Learning with Errors Problem (Invited Survey). In *Proceedings of the 2010 IEEE 25th Annual Conference on Computational Complexity*, CCC '10, page 191–204. IEEE Computer Society, 2010.

[34] R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, 1978.

[35] S. S. Roy, F. Vercauteren, N. Mentens, D. D. Chen, and I. Verbauwhede. Compact Ring-LWE Cryptoprocessor. In L. Batina and M. Robshaw, editors, *Cryptographic Hardware and Embedded Systems – CHES 2014*, pages 371–391, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg.

[36] P. W. Shor. Algorithms for quantum computation: Discrete logarithms and factoring. In *Proceedings 35th annual symposium on foundations of computer science*, pages 124–134. Ieee, 1994.

[37] K. Stoffelen. Efficient Cryptography on the RISC-V Architecture. In P. Schwabe and N. Thériault, editors, *Progress in Cryptology – LATINCRYPT 2019*, pages 323–340, Cham, 2019. Springer International Publishing.

[38] A. Waterman and K. Asanovic. The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Document Version 2.2. *RISC-V Foundation (May 2017)*, 2017.

Appendices

A. Register Map for All Optimized NTTs

RISC -V Register	NTT r1	INTT r1	NTT r2	INTT r2
a0	*input	*input	*input	*input
a1	*twiddles	*twiddles	*twiddles	*twiddles
a2	kyber_Q	kyber_Q	kyber_Q≪16	kyber_Q
a3	kyber_Q≪2	kyber_Q≪2	Qinv≪16	Qinv≪16
a4	Qinv	Qinv	*new_twiddles	Barrett const
a5	*new_twiddles	*psis_inv_montg	Barrett const	*new_twiddles
a6	value14	*new_twiddles	loop	loop
a7	value15	value15	omega	omega
t0	butterfly temp	butterfly temp	butterfly temp	butterfly temp
t1	butterfly temp	butterfly temp	butterfly temp	butterfly temp
t2	butterfly temp	butterfly temp	butterfly temp	butterfly temp
t3	value16	value16	value13	value13
t4	omega	omega	value14	value14
t5	value13	value13	value15	value15
t6	loop	loop	value16	value16
s0	value1	value1	value1	value1
s1	value2	value2	value2	value2
s2	value3	value3	value3	value3
s3	value4	value4	value4	value4
s4	value5	value5	value5	value5
s5	value6	value6	value6	value6
s6	value7	value7	value7	value7
s7	value8	value8	value8	value8
s8	value9	value9	value9	value9
s9	value10	value10	value10	value10
s10	value11	value11	value11	value11
s11	value12	value12	value12	value12
ra	Montg const	Montg const	kyber_Q	zeta_inv[127]
gp	-	value14	butter merge	kyber_Q≪16

Table A.1: Register Usage for NTT and INTT round 1 and 2

B. How to Reproduce the Results

INSTALLATION

1. Make sure that you install the **RISC-V GNU toolchain**. Follow the steps from the following link:

   `https://github.com/riscv/riscv-gnu-toolchain`

2. Make sure your current **jdk** is **1.8** (*java -version*). It does not work with newer versions of **jdk**. If your **jdk** version is newer, run the following commands:

   sudo apt-get install openjdk-8-jre

   sudo update-alternatives –config java

   Choose java 8 from the list generated in the last command.

3. Install SBT from: `https://www.scala-sbt.org/`

   For Ubuntu or Debian-based distribution, you can run the following commands:

   ```
   echo "deb https://dl.bintray.com/sbt/debian /" | sudo tee -a
   /etc/
   apt/sources.list.d/sbt.list
   ```

   `curl -sL "https://keyserver.ubuntu.com/pks/lookup?op=get`&*search= 0x2EE0EA64E40A89B84B2DF73499E82A75642AC823" | sudo apt-key add*

   ```
   sudo apt-get update
   ```

```
sudo apt-get install sbt
```

4. Install Verilator from: `https://www.veripool.org/wiki/verilator`
5. Clone the thesis project:

   ```
   git clone https://github.com/denigreco/Kyber_RISC_V_Thesis.git
   ```

6. Clone the pqriscv-vexriscv repository:

   ```
   clone https://github.com/mupq/pqriscv-vexriscv.git
   ```

HOW TO REPRODUCE THE RESULTS

First, go to the folder where the thesis project is cloned and compile all levels of Kyber using the following commands:

```
./build_everything.py -s pqvexriscvsim kyber512
./build_everything.py -s pqvexriscvsim kyber768
./build_everything.py -s pqvexriscvsim kyber1024
```

All the binaries are generated in the *bin* folder.

In order to reproduce the results you need to go to the folder where vexriscv is cloned and run the binaries one by one following the next instructions.

To get the results we are using the following template command where **crypto_kem_kyber512_kyber512r1_speed.bin** is the name of the binary (generated previously) we are executing:

```
sbt "runMain mupq.PQVexRiscvSim --init
../pqriscv/bin/crypto_kem_kyber512_kyber512r1_speed.bin"
```

Replace **kyber512** with **kyber768** or **kyber1024** to get the results specific to the other security levels of Kyber.

Replace **r1** with **r2** to get the results from round 2, and with **ref1** or **ref2** to get the results from the reference code of round 1 and 2. Each test has to be run individually.

The current command produces the **speed** results indicated by the last word in it: **speed**.

To compute how much Keccak takes in each block (key generation, encapsulation and decapsulation), replace the **speed** word with **hashing**.

To get the **code size** of Kyber round 1 and 2, go to the the thesis folder and then:

```
cd crypto_kem/code_size/round1
```

```
sh code_size.sh

cd ../round2

sh code_size.sh
```