Parallelise the NTT, you must

optimisations for the post-quantum key-exchange scheme NewHope-Simple for 64-BIT processors and a quick excursion to the LWR problem

Weenink, T.J.

Award date:
2017

Parallelise the NTT, you must

Optimisations for the post-quantum key-exchange scheme NewHope-Simple for 64-bit processors and a quick excursion to the LWR problem

Tim Weenink

Industrial and Applied Mathematics
Coding Theory and Cryptology
Technische Universiteit Eindhoven

Supervisors

Benne de Weger (Technische Universiteit Eindhoven)
Peter Schwabe (Radboud Universiteit Nijmegen)
Antonio de la Piedra (Compumatica Secure Networks)

Second reader

Jesper Nederlof (Technische Universiteit Eindhoven)

Technische Universiteit Eindhoven University of Technology

August 4, 2017

“Als ik zou willen dat je het begreep, had ik het wel beter uitgelegd.”

— Johan Cruijff

Acknowledgements

First of all, I would like to thank Benne, Peter, and Antonio for providing me with the excellent assistance during the entire period in which I have carried out my master project. Thanks to Jesper for being the second reader. Also, I'd like to thank COMPUMATICE SECURE NETWORKS[1] for giving me the opportunity to be part of their company and for allowing me to use all the material that was needed for this master project. Furthermore, I'd like to thank everyone who has supported me during my study. Thanks to my family and my friends. In particular, I'd like to thank the sweetest and most beautiful girl in the world, Maartje.

[1]https://www.compumatica.com/

Contents

List of Figures

List of Tables

List of abbreviations (LOA)

ALU	**A**rithmetic **L**ogic **U**nit
AVX	**A**dvanced **V**ector E**x**tensions
BCNS	**B**os **C**ostello **N**aehrig **S**tebila
BKZ	**B**lock **K**orkin-**Z**olotarev
CISC	**C**omplex **I**nstruction **S**et **C**omputer
CPI	**C**ycles **P**er **I**nstruction
CVP	**C**losest **V**ector **P**roblem
DFT	**D**iscrete **F**ourier **T**ransform
DSP	**D**igital **S**ignal **P**rocessing
FFT	**F**ast **F**ourier **T**ransform
FIPS	**F**ederal **I**nformation **P**rocessing **S**tandard
GCD	**G**reatest **C**ommon **D**ivisor
ISA	**I**nstruction **S**et **A**rchitecture
KEM	**K**ey **E**ncapsulation **M**echanism
LWE	**L**earning **W**ith **E**rrors
LWR	**L**earning **W**ith **R**ounding
MIPS	**M**icroprocessor with **I**nterlocked **P**ipeline **S**tages
NH	**N**ew**H**ope
NHS	**N**ew**H**ope-**S**imple
NIST	**N**ational **I**nstitute of **S**tandards and **T**echnology
NTT	**N**umber **T**heoretic **T**ransform
RISC	**R**educed **I**nstruction **S**et **C**omputer
RLWE	**R**ing-**L**earning **W**ith **E**rrors
RLWR	**R**ing-**L**earning **W**ith **R**ounding
SHA	**S**ecure **H**ash **A**lgorithm
SIMD	**S**ingle **I**nstruction **M**ultiple **D**ata
SVP	**S**hortest **V**ector **P**roblem

1 Introduction

Ever since the time of the ancient Greeks, people have been using cryptography in an attempt to communicate in a secure way [10]. Because of the increasing resistance against these security measures, the cryptographic methods have been developed in an increasingly advanced way. The arrival of the computer has given the opportunity to make the encryption and decryption protocols even more advanced, resulting in systems that are used nowadays to provide secure communication protocols, such as RSA [11], a protocol that relies on the hardness of factoring large numbers. In 1994 the American mathematician Peter Shor published an algorithm that is able to factor a large number into its prime factors in polynomial time, using a quantum computer [12]. For a long time, people have thought that it would be impossible to actually manufacture such a quantum computer, of which its tremendous computing power comes from its ability to perform multiple operations in parallel. Over the last few years, however, the quantum computer has made a spectacular rise [13], threatening all classic secure communication protocols. This is the reason that the field of post-quantum cryptography has gained a lot of attention lately [14].

In this master thesis we study the post-quantum key exchange mechanism NEWHOPE-SIMPLE (NHS) [2], show a way to speed up one of the protocol's main functions on 64-bit processors with a factor of almost 25% (and almost 40% on the MIPS64 architecture), and eventually propose a slight modification to the NHS protocol using the learning with rounding (LWR) assumption.

In Chapter 2 we will introduce the definitions that are necessary to understand the protocol and the underlying problems. We first describe the idea behind a quantum computer, together with Shor's algorithm. Afterwards, we introduce the concept of lattices and the corresponding problems that are supposedly difficult to solve, even with a quantum computer. Finally, we will describe the NEWHOPE (NH) protocol and the NHS protocol. In Chapter 3 we describe the processor that we have worked with during the course of this project. In Chapter 4 we discuss the strategies that have been used in order to optimise the protocol on our processor (and for 64-bit processors in general). Chapter 5 describes the LWR problem, the way it can be implemented in NHS instead of learning with errors (LWE), and the new failure probability and security analysis that it involves. The comparison between the original method and the optimised method can be found in Chapter 6. The discussion, proposals for future work related to this research, and the conclusion can be found in chapters 7, 8, and 9 respectively.

2 Preliminaries

In this chapter, we will make the reader familiar with the theorems and definitions that are needed to understand the problem and the goal of this thesis. In Section 2.1, we will introduce post-quantum cryptography by looking at quantum computers and Shor's algorithm. In Section 2.2 one can find background information on the lattices, the corresponding problems and their supposed hardness for quantum computers. Section 2.3 first explains the learning with errors problem and then extends this problem to the Ring-LWE (RLWE) variant. Finally, in Section 2.4 we introduce the post-quantum ephemeral key exchange protocol NH and its simplified version, NHS.

2.1 Post-quantum cryptography

In order to understand the idea of post-quantum cryptography, we will have to introduce the concept of quantum computers and their threat to the existing classical crypto.

2.1.1 Quantum computers

Today's computers are based on classical physics. Their memories consist of bits, the contraction of binary and digit, which has only 2 possible states: either 0 or 1. In most computing devices, such a bit is represented by an electrical voltage or current pulse. In 1982 the physicist Richard Feynman came up with the idea of using the effects of quantum mechanics in a computer, the so-called quantum computer, instead of using the classical mechanics [16].It makes use of qubits, short for **qu**antum **bits**, which have a very remarkable property. Besides of the existence in the states 0 and 1, it can also be in a superposition of both of these classical states. Say we have a physical system that can be in N different, mutually exclusive states. Following a notation designed especially for quantum states by Paul Dirac, the Dirac notation, we call these states $|1\rangle, |2\rangle, \dots, |N\rangle$. A quantum state $|\phi\rangle$ is a superposition of these classical states:

$$|\phi\rangle = \alpha_1 |1\rangle + \alpha_2 |2\rangle + \cdots + \alpha_N |N\rangle, \tag{1}$$

where α_i is called the amplitude of $|i\rangle$ in $|\phi\rangle$. A system in quantum state $|\phi\rangle$ is in state $|i\rangle$ with (probability) amplitude α_i, from which it follows that it is simultaneously in all classical states [17]. For example, a single qubit can be in a superposition of the 2 classical states 0 and 1. Likewise, a pair of qubits can be in any superposition of 4 classical states. If we continue this line of reasoning, we find that n qubits can be in all corresponding 2^n classical states simultaneously, whereas a classical computer can only be in 1 of these states at the same time. Combining this with other quantum-mechanical effects, such as interference and entanglement, we can build circuits with which we can compute multiple states in parallel [17].

2.1.2 Shor's algorithm

In 1994, the American mathematician Peter Shor published an algorithm for quantum computers that is able to find a factor of a composite number N that runs in a time polynomial in the input length $\log N$ [12]. Shor's algorithm goes as follows: For an odd composite input number N that has to be factor(is)ed, choose a random number $x < N$, such that they are coprime (otherwise we would have found a factor of N already). Next, compute $x^i \pmod N$ for $i = 2$ (because 0 and 1 are trivial) until we find some r such $x^r \pmod N = 1 = x^0 \pmod N$. This r is called the period. On page 29 of [17] it is shown that with probability $\geq 1/2$, r is even and $x^{r/2}+1$ and $x^{r/2}-1$ are not multiples of N. If that is indeed the case, we find

$$\begin{aligned} x^r &= 1 \pmod N && (2)\\ (x^{r/2})^2 &= 1 \pmod N && (3)\\ (x^{r/2}+1)(x^{r/2}-1) &= 0 \pmod N && (4)\\ (x^{r/2}+1)(x^{r/2}-1) &= kN \qquad \text{for some } k. && (5) \end{aligned}$$

Note that both terms are positive, so the k we find is positive as well. Also, because neither of the terms is not a multiple of N, both will share a non-trivial factor with N. In order to find this, simply compute the GCD of both terms with N. Should this have no useful factor as result, just pick another value for x and repeat until the factors have been found.

Now this might still seem as a computationally expensive algorithm at first sight. However, Shor found that finding the period r can be done efficiently with a quantum computer. In order to do so, he makes use of the quantum Fourier transform which runs extremely fast on a quantum computer. Another trick Shor uses is to apply repeated squaring in order to calculate modular exponentiation [17].

Using his algorithm, Shor showed that the supposedly hard problem of factoring large numbers is not as complex as it is for the classical computer, of which the latter formed the security basis of many well-known and widely used cryptosytems such as RSA. Combining this with the rise of the quantum computer, it it easy to understand that there is a huge need for new kinds of cryptosystems that resistant to quantum computer attacks. Of course, it can never be fully resistant to such a quantum computer attack, but it should be as hard for a quantum computer as it is for a classical computer. One kind of these so-called post-quantum cryptography systems is based on lattices. More information on this can be found in the next section.

2.2 Lattices

An n-dimensional lattice $\mathcal{L}$ is a discrete additive subgroup of $\mathbb{R}^n$. In order for $\mathcal{L}$ to be an additive subgroup of $\mathbb{R}^n$, it must contain the identity element $0 \in \mathbb{R}^n$, and $\forall x, y \in \mathcal{L}$ it must hold that their sum $x + y \in \mathcal{L}$ as well. Combining these two requirements, it follows that $-x \in \mathcal{L}$ as well.
Furthermore, discreteness means that for every lattice point $x \in \mathcal{L}$, one can construct an open ball $\mathcal{B}$ with radius $\epsilon > 0$ around x, denoted as $\mathcal{B}(x, \epsilon)$, such that $\mathcal{B}(x, \epsilon) \cap \mathcal{L} = \{x\}$. In other words, x is the only lattice point in a certain region around it. A simple example of a lattice can be found in Figure 1. The following definitions follow Peikert's notation.[19]

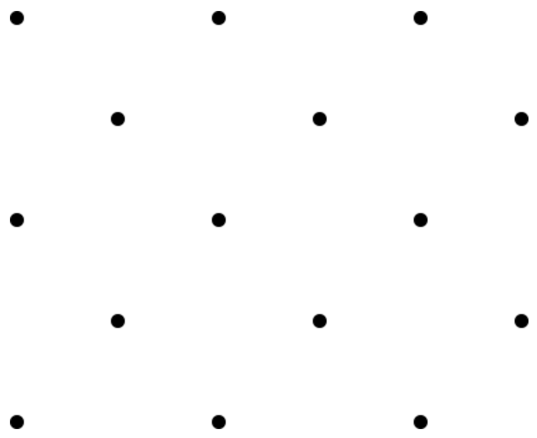


Figure 1: Part of a lattice in the 2-dimensional plane.

Definition 2.1. Lattice basis.
A basis $B = \{b_1, \ldots, b_n\} \subset \mathbb{R}^n$ of a lattice $\mathcal{L}$ is a set of linearly independent vectors whose integer linear combinations generate the lattice

$$\mathcal{L} = \mathcal{L}(B) := \Big\{\sum_{i=1}^{n} z_i b_i : z_i \in \mathbb{Z}\Big\}. \tag{6}$$

Note that such a basis is not unique, which forms the essential property of lattice-based cryptography. An example of a lattice basis is depicted in Figure 2.

Definition 2.2. Fundamental parallelepiped.
A lattice is some kind of structure of the fundamental parallelepiped of its basis, that keeps repeating in all n dimensions. The fundamental parallelepiped

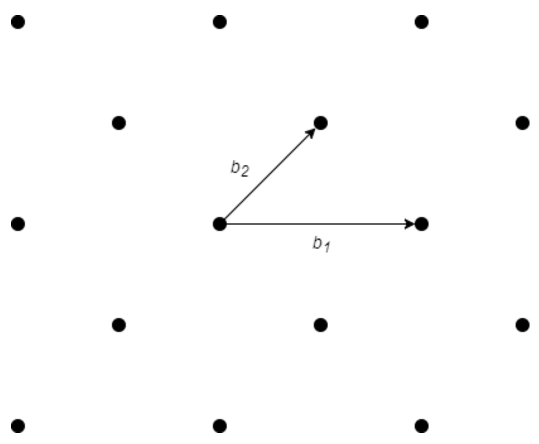


Figure 2: A basis of the lattice from Figure 1.

is defined as the parallelepiped that is spanned by the basis vectors, i.e.

$$\mathcal{P}(B) := B[0,1)^n = \Big\{\sum_{i=1}^{n} c_i b_i : c_i \in [0,1)^n\Big\}. \tag{7}$$

Constructing this area in 2 dimensions can be done by adding the tail of vector b_1 to the head of vector b_2 and vice versa. This is illustrated in Figure 3.

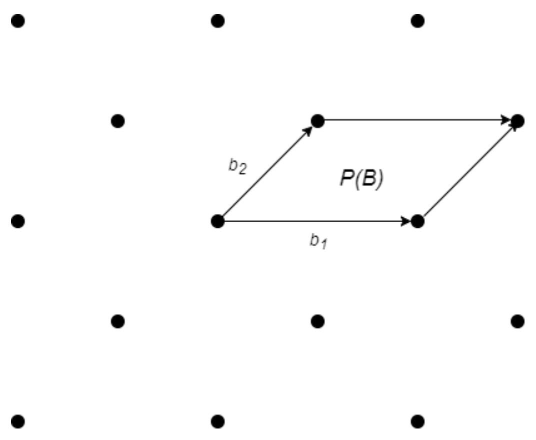


Figure 3: Fundamental parallelepiped of the lattice from Figure 1.

Note that this fundamental parallelepiped can also be constructed as the set of points that lie at most “half a vector” away from the origin, i.e.

$$\mathcal{P}(B) := B[-\frac{1}{2},\frac{1}{2})^n = \Big\{\sum_{i=1}^{n} c_i b_i : c_i \in [-\frac{1}{2},\frac{1}{2})^n\Big\}. \tag{8}$$

This is only a $(-\frac{1}{2}, -\frac{1}{2})$-translation of the previous definition. The volume of a lattice $\mathcal{L}$ is $\det(\mathcal{L}) = \det|B| = \mathrm{vol}(\mathcal{P}(B))$ for basis B.

Definition 2.3. Minimum distance of a lattice.
There is a vector $v \in \mathcal{L}$ that has minimal length. (Note that this vector is not unique). Then the minimum distance of a lattice $\mathcal{L}$ is defined as follows. The minimum distance of a lattice $\mathcal{L}$ is

$$\lambda_1(\mathcal{L}) := \min_{v \in \mathcal{L} \setminus v} ||v|| = \min_{x,y \in \mathcal{L}, x \neq y} ||x - y||, \tag{9}$$

where $||x||$ denotes the length of the vector x, calculated as the Euclidean distance.

Minkowski showed that for any lattice $\mathcal{L}$, we have that $\lambda_1(\mathcal{L}) \leq \sqrt{n}\det(\mathcal{L})^{1/n}$. For the proof, we refer to Peikert's lecture notes [19].

2.2.1 Ideal lattices

Both the NH protocol and the NHS protocol, that has been researched during this project, are based on the hardness of ideal lattice problems [1, 2]. The definition, given by Lyubashevsky et al. is as follows [15].

Definition 2.4. Ideal lattice.
Let $f \in \mathbb{Z}[x]$ be a monic polynomial of degree n. Consider the quotient ring $\mathbb{Z}[x]/\langle f \rangle$. An ideal lattice is an integer lattice $\mathcal{L}(B) \subseteq \mathbb{Z}^n$ such that $B = \{g \pmod{f} : g \in I\}$ for some ideal $I \subseteq \mathbb{Z}[x]/\langle f \rangle$.

When f is a monic, irreducible integer polynomial of degree n, every ideal I of the ring $\mathbb{Z}[x]/\langle f \rangle$ is isomorphic to a full-rank lattice in $\mathbb{Z}^n$. This lemma is also used in the proofs on the hardness.

Ideal lattices can also used in fully homomorphic encryption schemes. In 2009, Craig Gentry proposed the first solution of constructing such a scheme, based on ideal lattices [40].

2.2.2 Lattice problems

One of the questions that arise from the definitions of lattices is if there is a way to compute the minimum distance of a certain lattice or, even something that seems even more interesting, can we find a vector that actually has this minimum distance? This is known as the shortest vector problem (SVP). In the description of these problems, we use the notation from [20].

2.2.2.1 Shortest vector problem (SVP)

The SVP has 2 main variants: the search version and the decision version.

Definition 2.5. Shortest vector problem (search version).
Given a lattice basis B, find a vector $v \in \mathcal{L}$ (not equal to the all-zero vector) that suffices $||v|| = \lambda_1(\mathcal{L}(B))$.

This search version can also be extended to the approximate search version, which aims at finding a vector that has length $\gamma\lambda_1(\mathcal{L}(B))$ for some constant γ.

Definition 2.6. Shortest vector problem (decision version).
Given a lattice basis B and some positive $d \in \mathbb{R}$, determine whether $\lambda_1(\mathcal{L}(B)) \leq d$ or $\lambda_1(\mathcal{L}(B)) > d$.

The decision version is also known as GapSVP. Note that this function can also be extended to the approximate version, by simply adding the norm with a constant γ.

2.2.2.2 Closest Vector Problem (CVP)

The closest vector problem (CVP) is quite similar to the SVP we saw in the previous paragraph [55].

Definition 2.7. Closest vector problem (search version).
Given a non-zero vector $\mathbf{v}$ and a lattice basis B, find the lattice point $\mathbf{z} \in \mathcal{L}(B)$ that is closest to $\mathbf{v}$. That is, try to find the lattice point $\mathbf{z}$ that has distance

$$\text{dist}(v, \mathcal{L}(B)) = \min_{z \in \mathcal{L}(B)} \text{dist}(v, z) \tag{10}$$

from $\mathbf{v}$.

The decision version of CVP, known as GapCVP, is not about finding the actual closest lattice point.

Definition 2.8. Closest vector problem (decision version).
Given a non-zero vector $\mathbf{v}$ and a lattice basis B, determine whether or not there is a lattice point whithin a certain given distance d from $\mathbf{v}$.

Both version can also be approximated with a constant α, scaling the original problem with α.

2.2.2.3 Hardness of the lattice problems Problems would not be problems if they were easy to solve. In this section, we will show the difficulty of the previously mentioned lattice problems, together with some algorithms that can break these problems down.

The first conjecture of the hardness of a lattice problems dates from 1981, when Van Emde Boas proposed the conjecture that GapSVP belongs to the NP-hard problems [41]. In the same paper, he proved that the GapCVP belongs to the NP-hard problems. Almost 2 decades later Ajtai proved that Van Emde Boas' conjecture for the GapSVP holds for randomised reductions [42]. The approximation version of GapSVP, GapSVP$_\alpha$, is not NP-hard, however. Lenstra et al. introduced their famous LLL-algorithm, that, given a basis B with n-dimensional integer coordinates for a lattice $\mathcal{L}$, returns a short, nearly orthogonal lattice basis in time $O(d^5 n \log^3 b')$, where d is the number of basis vectors and b' is the length of the vector with the largest Euclidean distance to the origin [43]. This means that it is possible to approximate GapSVP$_\alpha$ in polynomial time within a factor $\alpha = O((\frac{2}{\sqrt{3}})^n)$, which is enormous for the parameters chosen in the NH(S) protocol. The idea behind the LLL-algorithm is to take the basis closest to the Gram-Schmidt vectors, improve the Gram-Schmidt vectors by changing their order. In 2001 Micciancio proved that the GapSVP$_\alpha$ stays NP-hard for $\alpha \leq \sqrt{2}$ [44]. Goldreich et al. showed that any algorithm that can efficiently approximate GapCVP$_\alpha$ can also efficiently approximate GapSVP$_\alpha$ [45]. The other direction is not known. Micciancio and Feige also showed that GapCVP$_\alpha$ stays NP-hard for $\alpha < \sqrt{\frac{5}{3}}$ [46].

Another famous lattice basis reduction algorithms is the block Korkin-Zolotarev (BKZ) algorithm, which basically reduces a lattice basis using an SVP oracle in a smaller dimension b [47, 1]. The authors of NH prove in Section 6.1 of their paper that this algorithm cannot solve the underlying lattice problems.

2.3 Ring-LWE

In this section we introduce the LWE problem, a famous lattice problem. Later, we will also discuss the LWE problem over a polynomial ring, which is known as the RLWE problem.

2.3.1 The learning with errors problem

The LWE problem was introduced by Oded Regev in 2005 [3]. It is a problem in machine learning, the area of computer science closely related to artificial intelligence, that is gaining much popularity lately. In his paper, Regev also gives a quantum proof that solving the LWE problem also implies an efficient algorithm to solve GapSVP$_\alpha$ for $\alpha = n\sqrt{n}$. Put differently, if one can efficiently solve LWE for $\mathbb{Z}_q^n$ on the average, then one can quantum-efficiently find short vectors in every n-dimensional lattice [3, 48].

Just like in the other lattice problems, the LWE problem can be split into multiple variants. The search version of the problem is as follows.

Definition 2.9. LWE (search version).
Given a function $f : \mathbb{Z}_q^n \to \mathbb{Z}_q$ that maps an n-dimensional vector x to a 1-dimensional vector y: $f(x) = y$. This function f is defined as taking the inner product of x with some fixed vector $s \in \mathbb{Z}_q^n$ and then adding a certain noise e, generated by a specific probability distribution, to it. You are given a number of samples of these pairs $(x, y) \in (\mathbb{Z}_q^n, \mathbb{Z}_q)$. Deduce the vector s from these samples.

An example of the search version, given by Regev in [21] is

$$14s_1 + 15s_2 + 5s_3 + 2s_4 \approx 8 \pmod{17} \tag{11}$$
$$13s_1 + 14s_2 + 14s_3 + 2s_4 \approx 16 \pmod{17} \tag{12}$$
$$6s_1 + 10s_2 + 13s_3 + 1s_4 \approx 3 \pmod{17} \tag{13}$$
$$10s_1 + 4s_2 + 12s_3 + 16s_4 \approx 12 \pmod{17} \tag{14}$$
$$9s_1 + 5s_2 + 9s_3 + 6s_4 \approx 9 \pmod{17} \tag{15}$$
$$3s_1 + 6s_2 + 4s_3 + 5s_4 \approx 16 \pmod{17} \tag{16}$$
$$\vdots \tag{17}$$
$$6s_1 + 7s_2 + 16s_3 + 2s_4 \approx 3 \pmod{17}. \tag{18}$$

This system of equations would have been easy to solve with Gaussian if the equations would have been exact. Instead, because of the (small) error that has been added to every equation, solving the system of equations becomes a lot harder. Note that the solution to this problem is $s = (0, 13, 9, 11)$.

Besides, there is also a decision variant of the LWE problem:

Definition 2.10. LWE (decision version).
Given a number of samples, distinguish the ones that have been generated by f from uniformly random samples.

Regev showed that the search version and the decision version are equivalent when q is bounded by some polynomial in n [3].

Going from search to decision is the most obvious reduction. Suppose we are given some kind of algorithm that can solve the search version of the problem. Now our challenge is to find out whether a range of pairs (x_i, y_i) are either input-output pairs of a certain function f, or uniformly randomly generated pairs. Feed the samples to the algorithm we use for the search version. If the output is a vector a for which all sample pairs hold, we know that they do not come from a uniformly random distribution. In case the samples are random, the algorithm will not be able to give an output vector a, so we can conclude that they are random.

Using the decision variant to retrieve the result of the search version goes a bit slower, but is still fairly easy to understand. Starting with the first coordinate, we would like to see what influence on y adding some constant r to the first coordinate of x_i would have. Say the input vector x_i represents $x_1, \ldots, x_n$ for a certain i. Then

$$(x; y) = (x_1, \ldots, x_n; \sum_{j=1}^{n} x_j s_j + e). \tag{19}$$

Adding $(r, 0, \ldots, 0)$ to the input vector x would result in

$$\begin{aligned}(\tilde{x}; \tilde{y}) &= (x_1 + r, x_2 \ldots, x_n; (x_1 + r)s_1 + \sum_{j>1}^{n} x_j s_j + e) & (20)\\ &= (x_1 + r, x_2 \ldots, x_n; y + rs_1). & (21)\end{aligned}$$

So we can make a guess k for s_1 and send the request $(x_1 + r, x_2 \ldots, x_n; y + rk)$ to the decision algorithm. If the algorithm tells us that the input pair indeed matches a function f, we have found the first coefficient of our vector s and we can repeat this trick for the next coefficient of s. In case our guess k was incorrect, the decision algorithm will tell us that it comes from a uniformly random distribution and we have to try another value of k [21].

The LWE problem is not quite useful in practice, since the key size is relatively very large. Because the secret vector needs to be multiplied with an entire matrix before adding the noise, we basically need n coefficients of this public matrix to get randomness for 1 position in the multiplied vector. This can be solved quite easily, by taking the negacyclic convolution of the public vector and the secret vector. This replaces the public matrix by a public vector, but because of the negacyclic convolution, all the coefficients are interconnected and thus the randomness is preserved. This is known as the Ring-LWE problem and is described in the following subsection.

2.3.2 Learning With Errors over a Ring (RLWE)

The follow-up of LWE is specialised to polynomial rings over finite fields and is called Learning With Errors over a Ring or Ring-Learning With Errors (RLWE). Again, let $\mathbb{Z}$ be the ring of rational integers and let $R = \mathbb{Z}[X]/(\Phi_m(X))$ be the ring of integers of the m^{th} cyclotomic number field. That is, $\Phi_m(X) \in \mathbb{Z}[X]$ is the m^{th} cyclotomic polynomial.

Definition 2.11. Cyclotomic polynomial.
The n^{th} cyclotomic polynomial is the unique irreducible polynomial with integer coefficients, which is a divisor of $x^n - 1$ and is not a divisor of $x^k - 1$ for any $k < n$. The roots of the cyclotomic polynomial are all n^{th} primitive roots of unity $e^{2\pi ik/n}$, where $\text{GCD}(k, n) = 1$ and $k < n$:

$$\Phi_n(x) = \prod_{\substack{1 \le k \le n \\ \gcd(k,n)=1}} \left(x - e^{2\pi ik/n}\right) \tag{22}$$

In case n is a power of 2, the cyclotomic polynomial will simply consist of the roots that correspond to the odd terms, since those are the only k's that are coprime to n:

$$\begin{aligned} \Phi_n(x) &= \prod_{1 \le k \le n \text{odd}} \left(x - e^{2\pi ik/n}\right) & (23) \\ &= \left(x - e^{2\pi i1/n}\right)\left(x - e^{2\pi i3/n}\right)\dots\left(x - e^{2\pi i(n-1)/n}\right) & (24) \end{aligned}$$

and since every pair of "mirrored" terms, i.e. the terms corresponding to root i and root $i + n/2$, cancel each other out, the result of this product will be $x^{n/2} + 1$.

In our case, m is a power of 2, so $\Phi_m(X) = X^n + 1$, where n is a power of 2 and $m = 2n$. We define R_q as $R/qR \cong \mathbb{Z}_q[X]/(X^n + 1)$, where $\mathbb{Z}_q$ represents the integers modulo q. Summarising, we see that we work with polynomials in X modulo X^n+1 where every coefficient is reduced modulo q.

Lyubashevsky et al. showed in [15] that the solution to this problem can be reduced to the supposedly NP-hard SVP in a lattice. Combining the latter with worst-case problems on ideal lattices being hard for polynomial-time quantum algorithms forms the security of post-quantum cryptography based on RLWE.

2.4 NewHope

The protocol that we have investigated during this project is NHS, a simplification of NH. These are post-quantum key-exchange schemes that are based on older key-exchange schemes. We start in 2012, when Ding et al. proposed a key encapsulation mechanism (KEM) that is based on the LWE problem [49]. Later, Peikert later proposed a simpler scheme, that also had better security claims than the scheme by Ding et al. [50]. This is based on the classical reduction from certain variants of the shortest vector problem to corresponding versions of the LWE problem, whereas first only a quantum reduction was known. In [51], Peikert proposed a passively secure KEM based on the RLWE problem. It makes use of a reconciliation mechanism in order to make sure that Alice and Bob agree on the same key, which ensures that no explicitly chosen key needs to be transmitted. The protocol can be found in Figure 4.

Alice		**Bob**
Generate $a \leftarrow \mathcal{R}_q$		
Generate $s, e \leftarrow \chi$		Generate $s', e', e'' \leftarrow \chi$
(Priv. key) s		(Priv. key) t
(Pub. key) $b = as + e$	$\xrightarrow{a, b}$	(Pub. key) $u = as' + e'$
		$v = bs' + e''$
		$\bar{v} \leftarrow \mathrm{dbl}(v)$
Decaps$(s, (u, v'))$	$\xleftarrow{u, v'}$	$v' = \langle \bar{v} \rangle_2$
$\mu \leftarrow \mathrm{rec}(2us, v')$		$\mu \leftarrow \lfloor \bar{v} \rceil_2$

Figure 4: Peikert's KEM scheme based on RLWE. For the actual details, we refer to [51].

Bos et al. proposed the Bos Costello Naehrig Stebila (BCNS) protocol, based on Peikert's scheme, that ensures 128 bits of post-quantum security [22]. In their protocol they choose to use lattices of dimension $n = 1,024$, with $q = 2^{32} - 1$. Also, they sample their errors from a discrete Gaussian distribution with parameter $\sigma = \frac{8}{2\pi} \approx 3.2$. The probability that Alice and Bob do not agree on the same key after reconciliation is $2^{-131,072}$, which is extremely small. One could say that this is maybe a bit too extreme.

Instead, the authors of NH(S) have decided to settle for less. In their protocol, they choose to use a centered binomial distribution, which is a lot easier to take samples from [1, 2]. Also, they transform the polynomials (using the NTT) in order to speed up the multiplication [1, 2]. Every run of the protocol, the seed a has to be refreshed to make sure that there can be no backdoors or all-for-the-price-of-one-attacks. Also, the error reconciliation is a slightly different from Peikert's method. Compared to BCNS, the number of cycles needed to perform 1 run of the protocol is a factor 8 or 9 smaller.

The NH scheme, as described in the original NH paper (in [1]) can be found in Figure 5.

Alice		**Bob**
Generate $\mathit{seed} \leftarrow \{0, \dots, 255\}^{32}$		
$\hat{a} \leftarrow \mathrm{Parse}(\text{SHAKE-128}(\mathit{seed}))$		
Generate $s, e \leftarrow \psi_{16}^{n}$		Generate $s', e', e'' \leftarrow \psi_{16}^{n}$
(Priv. key) $\hat{s} \leftarrow \mathrm{NTT}(s)$		
(Pub. key) $b \leftarrow \hat{a} \circ \hat{s} + \mathrm{NTT}(e)$		
$m_a = \mathrm{encodeA}(\mathit{seed}, \hat{b})$	$\xrightarrow{m_a}$	$(\hat{b}, \mathit{seed}) \leftarrow \mathrm{decodeA}(m_a)$
		$\hat{a} \leftarrow \mathrm{Parse}(\text{SHAKE-128}(\mathit{seed}))$
		(Priv. key) $\hat{t} \leftarrow \mathrm{NTT}(s')$
		(Pub. key) $\hat{u} = \hat{a} \circ \hat{t} + \mathrm{NTT}(e')$
		$v \leftarrow \mathrm{NTT}^{-1}(\hat{b} \circ \hat{t}) + e''$
		$r \leftarrow \mathrm{HelpRec}(v)$
$(\hat{u}, r) \leftarrow \mathrm{decodeB}(m_b)$	$\xleftarrow{m_b}$	$m_b = \mathrm{encodeB}(\hat{u}, r)$
$v' \leftarrow \mathrm{NTT}^{-1}(\hat{u} \circ \hat{s})$		
$\nu \leftarrow \mathrm{Rec}(v', r)$		$\nu \leftarrow \mathrm{Rec}(v, r)$
$\mu \leftarrow \text{SHA3-256}(\nu)$		$\mu \leftarrow \text{SHA3-256}(\nu)$

Figure 5: The original NH protocol [1]. Here $\circ$ denotes pointwise multiplication and every variable with a hat lives in the NTT domain.

2.4.1 The binomial distribution

One of the simplifications, and hence we can say improvements, with respect to the BCNS protocol ([22]) is the change of distribution of the LWE secret and error from discrete Gaussians by the centered binomial distribution ψ_k, $k = 16$. Not only is this distribution easier to implement, it is also protected against timing attacks [1]. One may sample from ψ_k by computing $\sum_{i=0}^{k} b_i - b'_i$, where $b_i, b'_i \in \{0, 1\}$ are uniform independent bits. This means that every bit follows a Bernoulli distribution (with probability $\mathbb{P}(b_i = 1) = \mathbb{P}(b_i = 0) = 1/2$). Suppose X is the sum of b_i and Y is the sum of $-b'_i$, $i = 1, \dots, k$, such that

$$X + Y = \sum_{i=0}^{k} b_i - b'_i. \tag{25}$$

Then $X = b_1 + \cdots + b_k$ is binomially distributed with parameters k $(= 16)$ and p $(=1/2)$, with mean kp $(= 8)$ and

$$\text{var}(X) = \text{var}(\sum_{i=0}^{k} b_i) \overset{b_i \text{ i.i.d.}}{=} \sum_{i=0}^{k} \text{var}(b_i) = kp(1-p) = \frac{k}{4}. \tag{26}$$

We know that $Y = -b'_1 - \cdots - b'_k = -(b'_1 + \cdots + b'_k)$, which is binomially distributed with mean $-kp$ $(= -8)$ and $\text{var}(Y) = \frac{k}{4}$, following the same reasoning as above. Since X and Y are independent, we see that the mean μ_{X+Y} and the variance σ^2_{X+Y} of their sum are simply given by $\mu_X + \mu_Y = 0$ and $\sigma^2_X + \sigma^2_Y = \frac{k}{2}$ respectively.

The way the bits b_i and b'_i are generated in the code is as follows. First of all, a random string t of length n with elements of 32 bits is generated using ChaCha20, a stream cipher by Bernstein [23]. Next, this string is shifted j to the right, $j = 0, \dots, 7$, and bit 1, 3, 5 and 7 are added to string d, which has length 32. Then a is given by the addition of the first and second byte of d and b is given by the addition of the third and fourth byte. So a represents the sequence of b_i and b represents the sequence of b'_i.

2.4.2 FIPS202

Another type of function that is often performed is the hash function from the Keccak family [52]. This is a family of hash functions, meaning that both the input as the output can vary in length. This family belongs to the Federal Information Processing Standards (FIPS) of the National Institute of Standards and Technology (NIST) by the American government [53]. The Keccak family includes both SHA3-256 and SHAKE-128, which are used in the NH(S) protocols.

2.4.3 Security of the NH(S) protocol

NH is a passively secure scheme. That means that the information that is sent from Alice to Bob can be intercepted eavesdropper Eve, without revealing any information that is useful to Eve. Eve can execute a passive attack by trying to retrieve information about the secrets from only the intercepted information. An active attack would also allow Eve to modify the messages that are sent from Alice to Bob and vice versa.

2.4.3.1 Man-in-the-middle-attacks

2.4.3.1.1 Passive attacks

Suppose that there is an eavesdropper, Eve, who intercepts the communication between Alice and Bob. This means that in the original protocol as proposed in NHS she finds the values

$$m_a = \text{encodeA}(\text{seed},\hat{b}), \tag{27}$$

and

$$m_b = \text{encodeB}(\hat{u}, \bar{c}). \tag{28}$$

Assume that Eve also knows the functions `decodeA`, `decodeB`, `Parse`, `SHAKE-128`, `NTT`, `NTT`$^{-1}$, `NHSEncode`, `NHSDecode`, `SHA3-256`, `NHSCompress` and `NHSDecompress` [1]. Using these functions, Eve can obtain the values of seed, $\hat{a}$, $\hat{b}$, $\hat{u}$, $\bar{c}$, c'. But due to the closest vector problem, finding the private keys s and s' from the known variables is extremely hard. Note that it is not possible either to obtain the shared secret ν just by eavesdropping. The only place where ν is communicated is in $\bar{c}$. But since s and t are unknown to Eve, retrieving ν comes down to solving the closest vector problem.

2.4.3.1.2 Active attacks

Instead of acting as a passive eavesdropper, Eve can also try to manipulate the variables that are sent between Alice and Bob. The most obvious way to manipulate the communicated variables is by sending a seed that consists of only zeroes, with the idea that Bob's private key will only be the error vector e'. However, Bob first uses SHAKE-128(seed) to obtain the basis $\hat{a}$. This means that Eve has to know the input vector such that its corresponding output vector using SHAKE-128 is all zero, which is extremely unlikely (if possible at all).

What's more, Eve can try to send the same seed multiple times in order to get a linear system of equations $u_i = at_i + e_i'$. Note that this will not work when both t_i and e'_i are changed every round. When t_i is the same in every round, let's say t, Eve can get a system $u_i = at + e'_i$. Keeping in mind that the e_i are binomially distributed, and when i is large enough, the total picture of all u_i will be centered around at. Since Eve knows a, this means she can also retrieve t. Note, however, that the amount of iterations i needed to get to this result will trigger an alarm on Bob's side.

Eve can also try to steal the shared secret ν by changing some variables. ν is only sent from Bob to Alice, and it's hidden in the variable $\bar{c}$. Eve can try to modify $\bar{c}$ by modifying the variable $\hat{b}$ that Alice sends to Bob. Still, $\bar{c}$ makes use of t, e'' and ν, which are all generated by Bob and thus unknown to Eve. The only malicious act Eve can perform is alter $\bar{c}$ such that Alice and Bob will disagree on the shared secret ν.

2.4.3.2 Malicious server or client

2.4.3.2.1 Malicious Alice (server)

Alice knows that Bob uses $\hat{b}$ in order to compute c, so that would be the only entrance to t. When she would set $\bar{b}$ as all ones, Bob's c will look like $t + e''k$. Alice can use this when she knows $e''k$, meaning that she can run the protocol incorrectly first, followed by a successful protocol run. Note that t, e'' and k should be the same for both runs in order for Alice to steal t.

2.4.3.2.2 Malicious Bob (client)

Bob only receives information from Alice, so his only entrance to Alice's private key is running multiple protocol runs. Alice changes the seed every time. Depending on whether or not Alice picks new values for s and e every time, he can run the protocol a number of times to create a linear system of equations and retrieve s.

2.4.4 Failure probability

The failure probability of the protocol is described in appendix D in [1]. It focuses on the size of the errors that can occur in the computation of v', i.e. the size of $et - e's + e''$. Because all these variables s, t, e, e', e'' come from the binomial distribution ψ_{16}, centered around 0, it is said to be σ-subgaussian, where $\sigma = \sqrt{8}$ [1]. Together with Lemma D.2, Lemma D.3 shows that even after the multiplication of all polynomials, with high probability the elements are small enough. Combining everything, Corollary D.4 concludes that, except with probability at most 2^{-61}, the error is at most 9,210, which is smaller than $\lfloor 3q/4 \rfloor - 2$. In Lemma C.3 it is shown that if this error is smaller than $(1 - 1/2^r) \cdot q - 2$, the reconciliation mechanism will give the same output for both v and v'. In NH, the r is chosen to be equal to 2 and $q = 12,289$, meaning that the error can be at most

$$(1 - 1/2^2) \cdot 12,289 - 2 = 3q/4 - 2, \tag{29}$$

which works for the distribution ψ_{16}.

2.5 NewHope-Simple

The simplified scheme, as described in NHS ([2]) can be found in Figure 6. The main difference with NH is that the reconciliation is left out. Instead of

Alice		**Bob**
Generate *seed* $\leftarrow \{0, \dots, 255\}^{32}$		
$\hat{a} \leftarrow$ Parse(SHAKE-128(*seed*))		
Generate $s, e \leftarrow \psi_{16}^n$		Generate $s', e', e'' \leftarrow \psi_{16}^n$
(Priv. key) $\hat{s} \leftarrow$ NTT(s)		
(Pub. key) $b \leftarrow \hat{a} \circ \hat{s} +$ NTT(e)		
$m_a =$ encodeA(*seed*,$\hat{b}$)	$\xrightarrow{m_a}$	$(\hat{b}$, *seed*$) \leftarrow$ decodeA(m_a)
		$\hat{a} \leftarrow$ Parse(SHAKE-128(*seed*))
		(Priv. key) $\hat{t} \leftarrow$ NTT(s')
		(Pub. key) $\hat{u} = \hat{a} \circ \hat{t} +$ NTT(e')
		$\nu \leftarrow \{0, 1\}^{256}$
		$\nu' \leftarrow$ SHA3-256(ν)
		$k \leftarrow$ NHSEncode(ν')
		$c \leftarrow \text{NTT}^{-1}(\hat{b} \circ \hat{t}) + e'' + k$
		$r \leftarrow$ HelpRec(v)
		$\bar{c} \leftarrow$ NHSCompress(c)
$(\hat{u}, \bar{c}) \leftarrow$ decodeB(m_b)	$\xleftarrow{m_b}$	$m_b =$ encodeB$(\hat{u}, \bar{c})$
$c' \leftarrow$ NHSDecompress$(\bar{c})$		
$k' = c' - \text{NTT}^{-1}(\hat{u} \circ \hat{s})$		
$\nu' \leftarrow$ NHSDecode(k')		
$\mu \leftarrow$ SHA3-256(ν')		$\mu \leftarrow$ SHA3-256(ν')

Figure 6: The NHS protocol [2]. Here $\circ$ denotes pointwise multiplication and every variable with a hat lives in the NTT domain.

retrieving the shared secret from the high bits of v, by removing the noise terms $et - e's + e''$, and applying the reconciliation function to agree on exactly the same key, we now generate a shared secret ν, encode 0 to 0 and 1 to $\lfloor q/2 \rfloor$, repeat the resulting string of integers 4 times, and add it to our message c. Alice computes k', and if this is done correctly, this will be an array of integers that are either around 0, which correspond to a 0 in the shared secret, or around $\lfloor q/2 \rfloor$, decoding to a 1. Note that we say *around*, because there can be (small) differences in the computation. We can find the element i in the decoded bitstring ν' by adding the values at positions i, $256 + i$, $512 + i$ and $768 + i$ in k'. If the sum is bigger than q, we put an 1 at position i in ν'. Otherwise, this translates to a 0.

A simplified version of this protocol, i.e. a version where we only focus on the core parts of the protocol, can be found in Figure 7.

Alice (server)		**Bob (client)**
Generate a $\in \mathbb{Z}_q^n, s, e \leftarrow \psi_{16}^n$		Generate $t, e', e'' \leftarrow \psi_{16}^n$
(Priv. key) s		(Priv. key) t
(Pub. key) $b = as + e$		
	$\xrightarrow{b, a}$	
		(Pub. key) $u = at + e'$
		Generate $\nu \in \{0,1\}^{256}$
		$k = \text{Encode}(\nu)$
		$c = bt + e'' + k$
	$\xleftarrow{u, c}$	
$k' = c - us$		
$= bt + e'' + k - (at + e')s$		
$= ast + et + e'' + k - ats - e's$		
$= k + et + e'' - e's$		
$\approx k$		

Figure 7: A simplification of the NHS protocol.

2.5.1 Failure probability

The difference in failure probability compared to the NH protocol is entirely dependent on the functions `NHSCompress` and `NHSDecompress`. In order to decrease the message size, each coordinate of c, which needs $\lfloor \log_2 q \rfloor + 1 = 14$ bits, is compressed to an integer in $[0, 7]$, needing $\lfloor \log_2 7 \rfloor + 1 = 3$ bits. The compression of an input value $x \in [0, q-1]$ works as follows. First, x is divided by q, such that $x \in [0, 1)$. Now, we multiply with 8, such that the output is in $[0, 8)$. After this, the value is rounded to the nearest integer. This implies that values exactly in the middle, i.e. values $j + \frac{1}{2}$, $j = 0, \dots, 7$, are rounded up. This implies that the values in $[0, 0.5)$ are rounded to 0 and the values in $[7.5, 8)$ are rounded to 8. In order to make `NHSCompress` uniform, we apply (mod 8) at the end, to map 8 to 0. Essentially, we divide the range $[0, q-1]$ into 8 intervals and simply tell in which of these interval the input coordinate lies. The decompression function tries to translate the intervals back to the original value by a multiplication with q/p. Note that this translates back to the centre of the intervals. This implies that the size of the compression error is at most half the size of an interval, so $\lceil \frac{1}{2} \cdot \frac{q}{8} \rceil = \lceil \frac{q}{16} \rceil$. Combined with Corollary D.4 [1], we see that the maximum error is, except with probability at most 2^{-61}, equal to $9,210 + 4 \cdot \lceil q/16 \rceil = 9,210 + 4 \cdot 769 = 12,286$, which is smaller than q.

In this chapter, we have described how the NHS protocol works, including all background information that is necessary. Furthermore, we have shown why it is passively secure and why it only fails with probability 2^{-61}. In Chapter 3, we will introduce the reader to the processor that has been used in this thesis to run, and optimise, the NHS protocol on. The properties of the processor are described and in Chapter 4 we describe optimisation techniques, based on these properties.

3 The MIPS64 processor

In this chapter we describe the processor that has been used during the course of this thesis. In Section 3.1 we give a short introduction to the processor's architecture. The concept of instructions is explained in Section 3.2. The way the instructions are handled on the processor is dealt with in Section 3.3. In Section 3.4 we explain how we can measure the number of cycles that are needed to perform a script. Our methodology is described in Section 3.5.

3.1 MIPS64 architecture

The processor that has been used during the course of this research is a MIPS64 Release 2 processor [5]. MIPS is a reduced instruction set computer (RISC) instruction set architecture (ISA) that has a set of attributes which gives the possibility of reducing the cycles per instruction (CPI) compared to a complex instruction set computer (CISC) [6]. A number of these instructions are described in Section 3.2. There are also several extensions to the ISA, such as the MIPS digital signal processing (DSP) [56], which will provide better arithmetic, and single instruction, multiple data (SIMD) [57], which can perform arithmetic operations more easily in parallel. In the release version that we have used during this project, these extensions were not available. However, we propose a SIMD approach in one part of NHS (the NTT) in Section 4.3.2.2.

During the 80's and 90's, MIPS processors have been used in personal, workstation and server computers by many companies. Also, MIPS is used in video game consoles, such as the Nintendo 64 and a range of Sony's PlayStations, and in the Tesla Model S [58].

MIPS64 makes use of 32 different registers, each of which is specified according to its intended purpose. The table with all available registers, together with their name and function, can be found in Table 1 and in [8].

Each of these registers contains 64 bits, which can contain one or multiple variables. For example, it can contain one doubleword of 64 bits, 2 words of 32 bits each, 4 halfwords of 16 bits each, or 8 bytes of 8 bits each. These bitstrings can be stored in (and loaded from) memory. However, these load and store instructions turn out to be cycle-consuming, which is what we have to keep in mind when we try to optimise (parts of) the protocol. In order to reduce the amount of cycles, we want to use as many registers as possible to do arithmetic operations with. Note that some of the registers, such as $at,

Name	**Number**	**Usage**
$zero	$0	Always 0
$at	$1	Reserved for assembler
$v0 - $v1	$2 - $3	Stores results
$a0 - $a3	$4 - $7	Stores arguments
$t0 - $t7	$8 - $15	Temporaries, not saved
$s0 - $s7	$16 - $23	Contents saved for use later
$t8 - $t9	$24 - $25	More temporaries, not saved
$k0 - $k1	$26 - $27	Reserved by operating system
$gp	$28	Global pointer
$sp	$29	Stack pointer
$fp	$30	Frame pointer
$ra	$31	Return address

Table 1: The available registers in the MIPS64 architecture.

$k0, and $k1, are reserved for specific purposes and can thus not be used in order to deal with calculations with variables. Furthermore, registers $zero and $sp cannot be modified, which leaves us with 27 registers to work with.

3.2 Instruction set

We would like to write code that can run on the MIPS64 processor. Therefore, we need to introduce the MIPS64 assembly language. This is a very low-level programming language, meaning that it is close to the machine language. For example, we can only apply operations on the registers that have been described in Table 1, instead of declaring variables and using them for computations such as in higher-level languages like C, Python, and Java. Furthermore, the number of instructions that we can use is quite limited. The entire instruction set for MIPS64 can be found in [7]. Each of these instructions is applied on at least 1 register. We can divide the instructions that are needed in this thesis into 5 categories:

First of all, we have the loads and stores. These instructions can load or store a byte/halfword/word/doubleword from or to a specific address in the memory. For example, the instruction `lh $t8, 0($a0)` loads the first halfword starting at position 0 of the input argument `$a0`'s address into register `$t8`. The offset, in this case equal to 0, denotes with how many bytes the address has to be shifted to the right in order to find the starting address.

The second category covers all the ALU instructions. This includes a range of instructions, from loading immediates into variables with `li` to instruction on 2 registers, such as addition, subtraction, AND, OR, and XOR. For example, the instruction `xor $t2, $t0, $t1` computes the XOR of the words in `$t0` and `$t1`, and stores the result in `$t2`. Note that some of these ALU instructions can also handle operations on doublewords. These instructions start with `D`, followed by the instruction name of the 32-bit equivalent. An example of this would be the instruction `DADD`, the addition of 2 doublewords.

Another category of which we use the instructions contains shiftings. Using `SLL $t9, 16` for example, we shift the bits in register `$t9` with 16 positions to the left. Note that this changes the register `$t9` permanently. Shifting an entire address can be done by adding the offset to the start of the address.

Multiplications and divisions have their own category. There is a clear distinction between multiplying or dividing signed integers and unsigned integers. In case we would like to use the latter, we simply add a `U` at the end of the `MULT` or `DIV` instruction.

The last category contains jumps and branches, which are necessary in loop environments. For example, the instruction `BLTZ` check whether or not the counter of a loop is less than or equal to zero. If not, we can decrement the counter with 1 and jump back to the start of the loop. Otherwise, if the final iteration is performed, it simply continues with the remainder of the code.

3.3 Pipeline structure

The MIPS64 processor is designed to be efficient. One of the methods that improve this efficiency is the pipeline structure that it uses in order to deal with multiple operations at a time. Instead of waiting for an instruction to finish in order to start with the new one, it is also possible to start the second instruction while the first one is still running. An example of this idea, where 4 words are being loaded, is depicted in Figure 8 [24].

3.4 Measuring the number of cycles

First of all, note that, due to technical issues, all the code has been simulated instead of actually run on the processor. We assume that the simulator returns the same output as the actual processor would give. In order to measure how fast our code runs, we measure the number of cycles that need to be performed on the processor. It turns out that we can simply read out

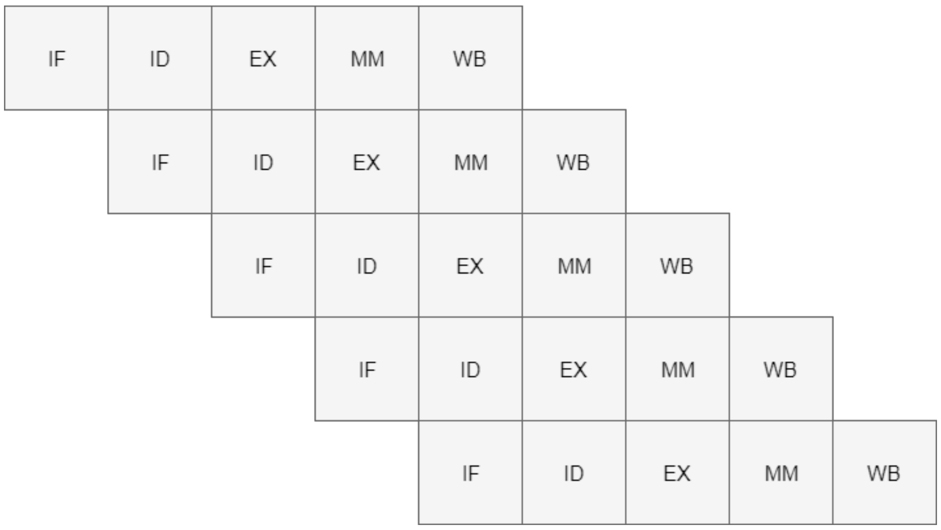


Figure 8: Example of the pipeline structure in MIPS64. Every row represents an instruction, every column represents a clock cycle. IF is fetching an instruction from memory, ID is decoding the instruction and reading the registers, EX is executing the operation or calculating the address, MM is accessing an operand in data memory, and WB is writing the results back into a register.

the `Count` register, of which the update frequency can also be provided as an input argument. This can be done by using the `RDHWR` instruction. Using this instruction, we are able to read the cycle counter of the processors as used in [9].

3.5 Methodology

The performance figures were obtained by cross-compiling for MIPS64 revision 2 architecture, using gcc version 4.7.0 compiler on a Debian Linux system for 64-bits with kernel 4.9.0. Afterwards, run the object code in our simulator in order to extract the required number of cycles and verify its functionality.

In Chapter 4 we will discuss strategies on how to use the MIPS64 processor, desribed in this chapter, in an optimal way in order to speed up the NHS protocol. In particular, we show how we can exploit the fact that our registers contain 64 bits to parallelise one of the methods that is used 6 times in the protocol.

4 Optimising NHS

In this chapter, we start in Section 4.1 by looking at related work to see whether there are optimisation strategies that we can copy for our own implementation. We then explain the techniques that are used in order to speed up the NHS protocol on the MIPS64 processor. In Section 4.2 we explain the reduction functions that have already been implemented in NHS, and show how they can be optimised for MIPS64. In Section 4.3, we describe the NTT and the corresponding optimisation strategies: the function we have optimised, not only in MIPS64 assembly, but also a general C implementation for 64-bit processors.

4.1 Related work

In the original NH paper ([1]), Section 7.3 is devoted to the optimised AVX2 implementation. This AVX2 is supported since Intel's Haswell generation and is capable of operating on 8 single-precision or 4 double-precision 256-bit vectors of integers of various sizes [26]. Güneysu et al. used the properties of the AVX2 in order to get a fast implementation (using only 4,480 cycles) for a dimension-512 NTT on the Intel Sandy Bridge processor [27]. In total, they managed to reduce the number of cycles needed to perform the NTT from 55,360 to 8,448 [1]. They also showed to speed up the noise sampling and the error recovery functions, which have been replaced in NHS.

The authors of NH also claim that, for certain parameter choices, their protocol can be implemented efficiently on small embedded processors. In [28], it is shown that this is correct, and an implementation on the ARM Cortex-M [29] family of 32-bit microcontrollers is presented. On the Cortex-M4[30], the NTT is sped up with almost a factor of 2: it takes 87,223 cycles to run the dimension-1024 NTT compared to 71,090 cycles on the dimension-512 NTT [28]. This number is achieved by merging multiple layers (at most 3 per 'chunk'), reducing the amount of loads and stores being called. This will be explained later in this chapter as well.

4.2 Reduction functions

NHS makes use of two famous reduction algorithms in order to speed up the computation. These are the Montgomery reduction and the Barrett reduction, which are used very often in NHS, making it interesting to optimise. These functions are described in the subsections below.

4.2.1 Montgomery reduction

The Montgomery reduction is a technique that has been invented by Peter Montgomery in 1985 [25]. It is used when a sequence of modular multiplications is computed, i.e. $c = ab \pmod{m}$, where m is fixed but a and b can be different every multiplication. The Montgomery reduction puts the numbers in a special form, the Montgomery form, allowing the multiplications to be more efficient than the ordinary methods. The algorithm is described in subsection 4.2.1.1. Afterwards, the C implementation and the MIPS64 assembly optimisation are described in subsection 4.2.1.2.

4.2.1.1 Algorithm

First, use the extended Euclidean algorithm in order to determine r^{-1} and m' such that

$$rr^{-1} - mm' = 1. \tag{30}$$

Next, convert the input arguments to Montgomery form:

$$\bar{a} = ar \pmod{m} \tag{31}$$
$$\bar{b} = br \pmod{m}. \tag{32}$$

Now, we would like to retrieve $u = abr \pmod{m}$. The easiest, and therefore also a quite expensive way to do this is to calculate $u = \bar{a}\bar{b}r^{-1} \pmod{m}$. The main reason why this is expensive is because of the need to calculate modulo m. Another way of calculating u, which may not seem very intuitive, but which can be much more efficient, is as follows:

$$t = \bar{a}\bar{b} \tag{33}$$
$$u = (t + (tm' \pmod{r}))m/r \tag{34}$$

If the latter is bigger than or equal to m, return $u - m$. Otherwise, simply return u. This equation is derived in the following way:

$$\begin{aligned}
t &= \bar{a}\bar{b} && (35)\\
u &= tr^{-1} \pmod m && (36)\\
&= trr^{-1}/r \pmod m && (37)\\
&= t(1 + mm')/r \pmod m \qquad \text{(because } rr^{-1} - mm' = 1\text{)} && (38)\\
&= (t + tmm')/r \pmod m && (39)\\
&= (t + tmm')/r + km \pmod m \qquad \text{(for any integer } k\text{)} && (40)\\
&= (t + tmm' + kmr)/r \pmod m && (41)\\
&= (t + (tm' + kr)m)/r \pmod m && (42)\\
&= (t + (tm' \pmod r))m)/r \pmod m && (43)
\end{aligned}$$

We can show that in the last line $(t + (tm' \pmod r))m)/r < 2m$, from which it is derived that the result is $u - m$ or u. If this would not hold, we would still have to reduce modulo m, which is the expensive step we wanted to avoid. We immediately see that $tm' \pmod r < r$. Furthermore, $\bar{a}, \bar{b} < m$, so $t < m^2$. Combining the results, we find

$$(t + (tm' \pmod r))m)/r < (m^2 + (rm))/r = m^2/r + m. \qquad (44)$$

Since $r > m$, we see that this is indeed smaller than $2m$. So instead of calculating modulo m, we have to calculate modulo r and divide by r. Although the latter might seem more memory- and time consuming, we can show that it is not by making use of the fact that r is a power of 2. Calcuting the remainder of a number divided by r can simply be performed by taking the last $\log_2(r) + 1$ bits of the number. Also, dividing by r can be done efficiently by shifting the number $\log_2(r)$ bits to the right.

The last step consists of converting the result from Montgomery form to normal form by calculating

$$ab = ur^{-1} \pmod m. \qquad (45)$$

4.2.1.2 Implementation

The implementation of the Montgomery reduction in C can be found below. Note that the input argument a is $\bar{a}\bar{b}$.

```
uint16_t montgomery_reduce(uint32_t a)
{
    uint32_t u;
    u = (a * qinv);
    u &= ((1<<rlog)-1);
    u *= PARAM_Q;
    a = a + u;
    return a >> 18;
}
```

First, it calculates tm' and then it determines the remainder after division by r by taking the last $\log_2(r) + 1$ bits with a logical `and`. After that, the result is multiplied by m and it is added to the input argument. The bit shift in the last line represents division by r.

In assembly, it is optimised in the following way:

```
montgomery_reduce_asm:
    li      $t0, 12287
    mul     $v1, $v0, $t0
    ext     $v1, $v1, 0, 18
    li      $t0, 12289
    mul     $v1, $v1, $t0
    add     $v1, $v1, $v0
    ext     $v0, $v1, 18, 14
    jr      $ra
```

Note that this is the implementation of the separate function. When the Montgomery reduction is used inside the NTT, there is no need to load the values $12,287$ and $12,289$ every time the loop is called. Instead, we can load these constants outside of the loops, meaning that the load instructions in the assembly code above can be omitted, reducing the amount of cycles needed in order to perform the function on the MIPS64 processor by 2.

4.2.2 Barrett reduction

In 1986, Paul Barrett introduced an algorithm to compute $c = a \pmod{n}$ for a constant n [31]. The idea behind this algorithm is described in Subsection 4.2.2.1. The C implementation and the corresponding optimisation in assembly can be found in subsection 4.2.2.2.

4.2.2.1 Algorithm

The easiest, most obvious way of computing $c = a \pmod{n}$ would be to subtract n from a a couple of times and stop whenever the result is smaller than n. This is equivalent to calculating $c = a - kn$, where $k = \lfloor \frac{a}{n} \rfloor$. Because division can be expensive, and the same n is used every time, Barrett invented a way of approximating n's inverse, $\frac{1}{n}$, which makes use of the binary structure:

$$\frac{1}{n} = \frac{2^i/n}{n(\frac{2^i}{n})} = \frac{2^i/n}{2^i} = \frac{m}{2^i}, \tag{46}$$

for a certain i. We'd like to use an integer value for m, which means that $\frac{2^i}{n}$ has to be rounded. Because it is not allowed to subtract more than $\frac{1}{n}$ times an from a, the obvious solution is to choose $m = \lfloor \frac{2^i}{n} \rfloor$. (Note that the value of i has to be at least $\lceil \log_2(n) \rceil$ in order to be useful. In the case of NHS, $i = 16$ is used.) In NHS, the constant value $n = 12,289$ is used. Together with $i = 16$, we find that the value of m is $\lfloor \frac{2^{16}}{12289} \rfloor = 5$. So the Barrett reduction needs some precomputation, which is actually quite trivial to do, in order to reduce the calculation cost.

4.2.2.2 Implementation

The implementation in C is as follows:

```
uint16_t barrett_reduce(uint16_t a)
{
    uint32_t u;
    u = (uint32_t) a;
    u = u * 5;
    u = u >> 16;
    u = u * 12289;
    a = a - u;
    return a;
}
```

Precomputation of the constants yields $m = 5$ and $i = 16$. The operation `u = u >> 16` means shifting the bitstring `u` with 16 positions to the right, which is the same as dividing u by 2^{16}. In assembly it is implemented as:

```
barrett_reduce_asm:
    li      $v1, 5
    mul     $v0, $a0, $v1
    srl     $v0, $v0, 0x10
    li      $v1, 12289
```

```
        mul     $v0, $v0, $v1
        sub     $v0, $a0, $v0
        jr      $ra
```

Similar to the implementation of the Montgomery reduction, we can load the constants outside the loop. This would save 2 cycles every time the Barrett reduction is called.

4.3 Number theoretic transform (NTT)

One of the building blocks of NH(S) is the NTT, as can be seen in tables 3 and 5. As we see in Figure 6 the NTT is called 6 times in the entire NH(S) protocol, with 2 regular NTT calls and 1 inverse NTT on either side. Because it is used so often, a significant optimisation of the NTT would directly imply a significant optimisation for the entire protocol.

In this section, we will first give a brief introduction to Fourier transforms, which provide a fast way of polynomial multiplication. Later, we will explain the NTT and its structure. Keeping this structure in mind, we will describe optimisation strategies that exploit it.

4.3.1 Fourier transforms

In order to understand other methods of polynomial transformation, we need to introduce the concept of (primitive) roots of unity. (These definitions are taken from [34]).

Definition 4.1. Root of unity.
A number ω is an n^{th} root of unity if $\omega^n = 1$, $\omega \in \mathbb{C}$.

Definition 4.2. Primitive root of unity.
A number ω_n is a primitive n^{th} root of unity if it is a root of unity and if, in addition, n is the smallest integer of $k = 1, \dots, n$ for which $\omega_n^k = 1$.

As mentioned in [32], there are exactly n complex n^{th} roots of unity, which are given by formula (47):

$$\omega_n^k = e^{2\pi ik/n} \text{ for } k = 0, 1, \dots, n-1. \tag{47}$$

Using Euler's formula, this can be rewritten as

$$\omega_n^k = \cos(2\pi k/n) + i\sin(2\pi k/n) \text{ for } k = 0, 1, \dots, n-1. \tag{48}$$

For example, the complex 4th roots of unity are repesctively 1, i, -1, and $-i$, which are exactly the intersections of the unit circle with the axes of the complex plane $\mathbb{C}$.

Regular multiplication, also known as schoolbook multiplication, of two $n-1$-degree polynomials needs $O(n^2)$ operations. This number can be reduced to $O(n \log n)$ by working with the Fourier transforms of the polynomials. Let us first introduce the definition of the Fourier transform.

Definition 4.3. Fourier transform.
The Fourier transform of a function $x(t)$ is given by

$$X(\omega) = \int_{-\infty}^{\infty} x(t)e^{-i\omega t}dt. \tag{49}$$

The corresponding inverse Fourier transform is given by

$$x(t) = \frac{1}{2\pi}\int_{-\infty}^{\infty} X(\omega)e^{i\omega t}d\omega. \tag{50}$$

Here $x(t)$ represents a function in time and $X(\omega)$ stands for a function of frequency. The Fourier transform has the nice property that multiplying 2 transformed polynomials and taking the inverse Fourier transform of their product gives the correct result. This is known as the convolution theorem. Expressed as a formula, the convolution of $X(\omega)$ and $Y(\omega)$ is

$$X * Y = \int_{-\infty}^{\infty} Y(\omega')X(\omega - \omega')d\omega'. \tag{51}$$

The proof of the convolution theorem can be found in [38].

Since we work with integer polynomials, it makes more sense to introduce the discrete Fourier transform (DFT).

Definition 4.4. Discrete Fourier transform (DFT).

$$y_k = f(\omega_n^k) = \sum_{j=0}^{n-1} x_j\omega_n^{jk}, \tag{52}$$

for $k = 0, 1, \ldots, n-1$, where ω_n is the n^{th} root of unity.

This formula is similar to the one in equation (50). In essence, the x is a sum of the functions on smaller intervals. The inverse of the DFT is defined as

$$x_j = \frac{1}{n}\sum_{k=0}^{n-1} y_k\omega_n^{-jk}, \tag{53}$$

where $k \in [0, n-1]$. The DFT gets interesting when the number of input coefficients is a power of 2, because it allows us to construct a binary tree on which we can apply a divide-and-conquer method.

The fast Fourier transform (FFT) splits the coefficients into 2 blocks of equal size every layer, and performs a so-called butterfly operation on a pair of coefficients, each coming from a different block. An example of the FFT of a polynomial with 32 coefficients is given in Figure 9. Note that the output is bit-reversed. For example, the second row has input coefficient x_1, which has binary representation 00001_2, and output coefficient X_{16}, which has binary representation 10000_2, the mirrored version of x_1. Each connection of 2 coefficients is called a butterfly, because of its shape. In NHS, the Gentleman-Sande butterfly is used. This type of butterfly is depicted in Figure 10. Now these butterflies are still performed using a complex root of unity ω_n. Instead of working with the complex numbers, we can also determine the transforms over a finite field $F = \mathbb{Z}_p$. This type of transformation is known as the number theoretic transform (NTT). From Fermat's little theorem, we know that for prime p and any integer a:

$$a^p \equiv a \pmod{p}, \tag{54}$$

and from this it follows that

$$a^{p-1} \equiv 1 \pmod{p}. \tag{55}$$

There exists a primitive root r, such that the results of calculating $r^i \pmod{p}$ form a permutation of the elements $0, \ldots, p-1$ for $i = 0, \ldots, p-1$. We would like to have $\omega^n = 1 \pmod{p}$, where n is the length of the resulting polynomial. For this, we need a prime for which the relation $p-1 = kn$ holds for some k, so $p = kn + 1$. Next, we find the primitive root r and calculate ω as $r^k \pmod{p}$. This works, because

$$\omega^n = r^{k*n} = r^{p-1} = 1 \pmod{p}. \tag{56}$$

Because r is the primitive root mod p, it is clear that for a power $n' < n$ the result of $\omega^{n'}$ will not be equivalent to $1 \pmod{p}$.

4.3.2 Optimisation strategies

In the following paragraph we will describe different strategies that can be applied in order to calculate the number theoretic transform of a certain polynomial of degree 1,023, i.e. a polynomial with 1,024 coefficients. We first describe the way the NTT is implemented in the reference C implementation of NHS. Afterwards, we will describe methods that are supposed to reduce the amount of cycles needed to perform the NTT. Some of these methods only work in theory. In order to make them work in practice as well, we have to introduce some slight modifications. This is explained at the end of this subsection.

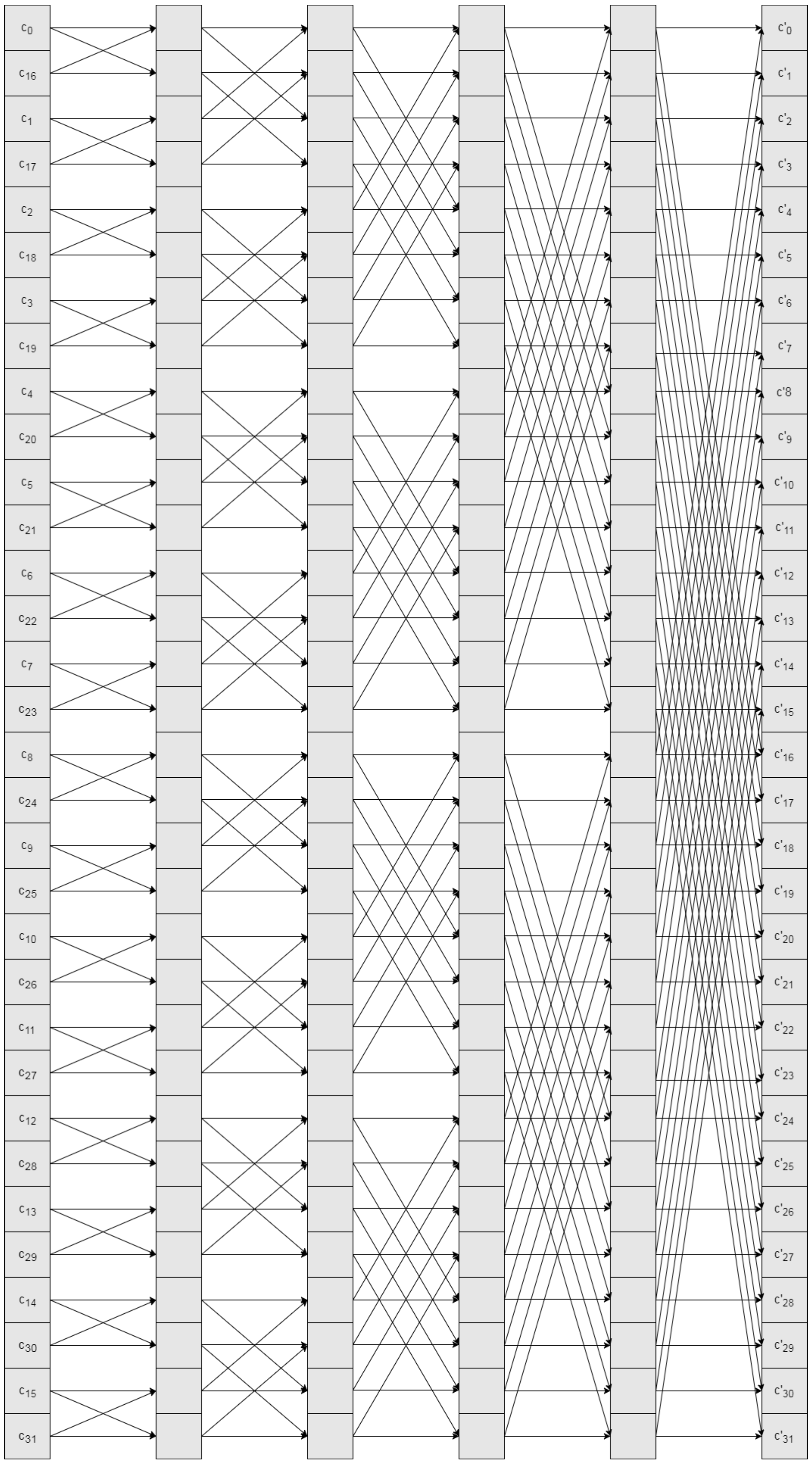


Figure 9: Calculation of the FFT for a polynomial with 32 coefficients.

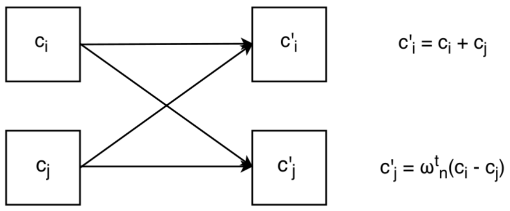


Figure 10: The Gentleman-Sande butterfly of the coefficients c_i and c_j.

For the moment we assume that we can use all 32 registers for the calculation, loading and storing of the coefficients and the intermediate values. Each register contains 64 bits, which means that we can store 4 coefficients per register. Our main goal is to reduce the amount of `load` and `store` instructions, since they are by far the most costly operations. That does not mean that we are not willing to take other optimisations into account.

4.3.2.1 The naive way

The most obvious and therefore also the most costly way to compute the NTT is to work from top to bottom, from left to right. Since every computation is done using 2 coefficients, 1 register (or actually even half of it) would suffice to perform the entire transformation. Note that this would, however, use 1,024 loads and 1,024 stores per layer, so 10,240 loads and 10,240 stores for the entire NTT. Note that this only concerns the loading and storing of the coefficients. In practice, we would also need to load the constants, such as the ωs, in order to calculate the NTT correctly.

4.3.2.2 The parallel method

Instead of only using half a register, as desribed in Section 4.3.2.1, we could, in theory, also make use of all the registers. That is, we could load 4 coefficients in every of the 32 registers, so using 128 coefficients at a time, perform the butterfly operations in parallel and store all 128 coefficients again. For example, we can store the first 4 coefficients of c in a register. This is depicted in Figure 11. In the ideal case, we can perform one butterfly operation

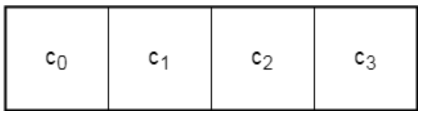


Figure 11: A register of 64 bits containing 4 coefficients of 16 bits each.

for 4 pairs of coefficients simultaneously by performing the operation on 2 entire registers. Afterwards, we could retrieve the individual coefficients by applying bit masks that only select the halfword that we want. For example, c_1 can be found by applying the mask `0x0000FFFF00000000` to the register in Figure 11. This technique, based on using basic instructions to perform SIMD operations, was proposed in [54]. This would mean there is only need for 2 multiplications to determine 8 new coefficients, which is only a quarter of the number of operations that would be needed in the most naive way. Parallelisms like this can also significantly reduce the number of cycles. Note, however, that the amount of loads and stores is still the same as in the most naive way. An example of the parallelised version is depicted in Figure 12.

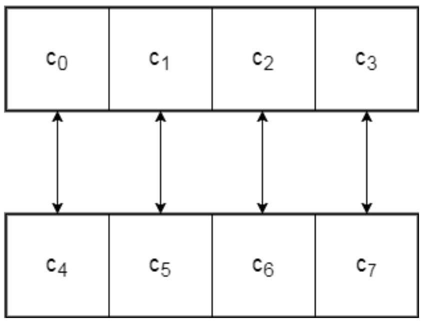


Figure 12: An example of the (theoretical) parallel method applied on the first 8 coefficients of c. Each arrow denotes a butterfly operation.

4.3.2.3 Pendulum

A slight improvement to the parallel method can be made by not working from top to bottom in every layer but by alternating the direction in which the operations are performed. Let us illustrate this as follows. Suppose we start at the top. Then we perform the butterfly operations pairwise to the coefficients in the current block that contains 128 of those. After that, we proceed to the 2nd block from the top and continue. Eventually, the 8th, and thus last, block has been handled and the results of the corresponding butterfly operations are still in the registers. Then we move to the next layer and start from the bottom, meaning that we can proceed with the numbers that are still in the registers, and so saving the costs of the stores and loads of these coefficients. Similarly, jumping from the 2nd to 3rd layer can be done without storing and loading the upmost block, and so on. Using this method, you can save $9 \cdot 128 = 1,152$ loads and 1,152 stores when applied to the parallel method.

4.3.2.4 Blockwise method

A more sophisticated variant consists of finishing an entire sequence of butterfly operations in a block of (adjacent) coefficients before proceeding to the next block. Because of the bit reversal that is done before entering the actual loop, all coefficients are correctly aligned for the butterfly operations. In the first layer, for example, all butterfly operations are done between two adjacent coefficients. Also, every butterfly needs a different ω, so there is a clear distinction between the 512 butterflies.

In the next layer, the distance gets doubled, and every ω is used twice in a row. In other words, every ω corresponds to a block of 4 coefficients and 2 butterflies. This means we now have to perform deal with 256 blocks of 4 coefficients. Continuing like this, we find that in layer 7 there are 8 blocks of 128 coefficients, which is exactly how many coefficients we can theoretically handle at once. From layer 8 onwards, the groups in which butterfly operations take place outgrows the 128-coefficient blocks. This can be solved by performing the last 3 layers naively. Otherwise, we can rearrange the coefficients and apply the blockwise method again. A depiction of the computations in one such block can be found in Figure 9.

For the first 7 layers we can split the coefficients into 8 blocks of 128 and work block by block. Starting with the upper block (with coefficients $a_0 \dots a_{127}$) in the first layer, we can calculate the results of the butterfly operations and overwrite the original values. We then move one layer to the right, using the same registers with the newly calculated values (meaning we can simply skip the load and store part) to perform the butterfly operations on. This process can be repeated up to and including layer 7, where the results have to be stored and we can move to the second block (with coefficients $a_{128}, \dots, a_{255}$) of the first layer. Once we have done this for all 8 blocks, we finish the algorithm by performing the last 3 layers. This method uses $8 \cdot 128 = 1,024$ loads and $1,024$ stores for the first 7 layers and then (using the naive mode) $3 \cdot 8 \cdot 128 = 3,072$ loads and $3,072$ stores in the last 3 layers, or $8 \cdot 128 = 1,024$ loads and $1,024$ stores by rearranging and applying the blockwise method. This results in $1,024 + 3,072 = 4,096$ loads and $4,096$ stores for the former and $2 \cdot 1,024 = 2,048$ loads and $2,048$ stores in the latter. Note that we can also apply the pendulum method, saving even more loads and stores.

4.3.2.5 Removing the dummy

In the original butterfly computation, the old coefficients have to be replaced by the new values. In order to do so, a new dummy value is introduced, such that the old values are not overwritten until the end of the butterfly. Say that we want to apply a butterfly on the coefficients a and b, then we get

$$c = a \tag{57}$$
$$a = a + b \tag{58}$$
$$b = \omega(c - b). \tag{59}$$

Rewriting these expressions, we find out there is actually no need to introduce such a dummy variable:

$$\tilde{a} = a + b \tag{60}$$
$$\tilde{b} = \omega(a - b) = \omega((a + b) - 2b) = \omega(\tilde{a} - 2b), \tag{61}$$

where $\tilde{a}$ and $\tilde{b}$ can simply overwrite the original a and b. The only difference with the previous solution is that we have to use an extra instruction (the computation now involves $-2b$ instead of $-b$). This can be done by either multiplying b by the immediate 2 first and then subtracting it, or simply subtracting b twice. On the other hand, we only need 1 extra register in total for the entire NTT.

4.3.2.6 From theory to practice

As we have already seen in Table 1, we cannot make use of all 32 registers in practice. In fact, we can only use 27 registers. Since we also need to use quite a lot of them for other aspects than the coefficients, it turns out that we have to use 11 registers for other purposes, leaving 16 registers for the coefficients. Also, because of the overflow bits that are needed for the addition and the multiplication in the butterfly, in practice we can only store 2 coefficients in 1 register. The way they are stored, is depicted in Figure 13. This means the new amount of layers we can merge, i.e. process at the

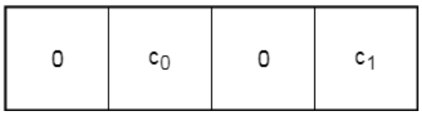


Figure 13: The way the coefficients are stored in 1 register. The 0-blocks consist of 16 zeros and are reserved for the overflow bits of the coefficients c_0 and c_1.

same time without having to store in between, has to be calculated again. Instead of 128 coefficients, we can "only" deal with $2 \cdot 16 = 32$ coefficients at a time. That means we can merge at most $\log_2 32 = 5$ layers. In our

MIPS64 assembly optimisation, we choose for merging at most 4 layers. The reason behind this is the following. First of all, the conversion of 16-bit input coefficients to 64-bit coefficients in the first layer has to be handled separately. Afterwards, we could theoretically merge 5 layers. However, after the 5th layer, the blocks of coefficients have to be chosen differently than simply choosing a block of adjacent coefficients. From the 6th layer onwards, one coefficient block is formed by choosing every 32nd coefficient (and its direct neighbour). Using this method, we can merge the 6th, 7th, 8th, and 9th layer. Similar to the first layer, the last layer is handled separately in order to convert the output to 16-bit integers again.

4.3.2.7 Reduction functions for 64-bit input

When we switch from 1 coefficient per register to 2 coefficients, we also have to modify some other functions in order to receive the correct output. For example, inside the loop of the NTT there are several calls to the Barrett reduction and the Montgomery reduction, which are originally not designed to deal with 64-bit integers. For example, in the Barrett reduction there is a line in which a shift to the right with 16 bits takes place. Using 32-bit integers, this implies deleting the 16 least significant bits, shifting the 16 most significant bits to the places where the least significant bits used to be, and filling up the 16 significant bits with all zeroes. The latter would not work in case the operation would take place on a 64-bit integer containing 2 coefficients: shifting the leftmost coefficient would move its 16 least significant bits to the 16 most significant bits of the rightmost coefficient. This can be solved easily, by applying a mask to the coefficients before performing the bit shift. This masking makes sure that the 16 least significant bits of both coefficients are set equal to 16 zeroes. The C code is given below.

```
uint64_t barrett_reduce64(uint64_t a)
{
    uint64_t u;
    u = a;
    u = u * 5;
    u = u & 0xFFFF0000FFFF0000;
    u = u >> 16;
    u = u * 12289;
    a = a - u;
    return a;
}
```

The modified assembly code can be found in the Appendix B.

Being able to guarantee the functioning of the Montgomery reduction for 64-bit integers is slightly trickier. Intuitively, we can expand the original function with a mask on 2 places in the code. The C code of this naive idea is given below.

```
uint64_t montgomery_reduce64_naive(uint64_t a)
{
    uint64_t u;
    u = (a * qinv);
    u = u & 0x0003FFFF0003FFFF;
    u *= PARAM_Q;
    a = a + u;
    a = a & 0xFFFC0000FFFC0000;
    return a >> 18;
}
```

Experimenting with this function, one finds out that it still contains a flaw causing a small mistake in the coefficient in the higher bit. After investigating the intermediate steps, we find out that the multiplication of a and q^{-1} in the first line does not return the desired result. Multiplying the coefficient in the lower bits with q^{-1} results in an integer of which the representation also needs some of the higher bits, i.e. some of the bits that are actually intended for the other coefficient. This so-called overflow can be dealt with by handling the coefficients separately and adding them later, at a point where they cannot influence each other. In order to do so, we need to introduce at least one new variable, $u1$. For the sake of readability, we choose to also use $u2$.

```
uint64_t montgomery_reduce64(uint64_t a)
{
    uint64_t u, u1, u2;
    u1 = a & 0xFFFFFFFF00000000;
    u1 *= qinv;
    u2 = a & 0x00000000FFFFFFFF;
    u2 *= qinv;
    u1 &= 0x0003FFFF00000000;
    u2 &= 0x000000000003FFFF;
    u = u1 + u2;
    u *= PARAM_Q;
    a += u;
    a &= 0xFFFC0000FFFC0000;
    return a >> 18;
}
```

4.4 Other assembly optimisations

Not only are we interested in the particular functions of the NTT. We can also optimise the assembly code (slightly) by looking at general optimisation tricks.

4.4.1 Unrolling the loops

One of those is unrolling the loops. Usually we would define the body of a loop, a counter that is incremented at the end of the loop, and a line in which it is checked whether the loop should be run again. Jumping to the start of the loop takes a lot of cycles. These cycles can be removed by unrolling the loop. That is, manually increment the counter (if necessary) and duplicate the body of the loop as many times as the loop has to be performed. This has the disadvantage that the code will get really long. In assembly, this is solved by making use of macros. We can define macros that contain code. When we want to use this code, we only have to state the name of the macro (and the input parameters). The compiler automatically copies the macro body on the places where the macro is called. In MIPS, a macro looks like

```
.macro EXAMPLE
   BODY
.endm
```

Each time `EXAMPLE` is called in the code, the compiler replaces it with `BODY`.

4.4.2 Binary multiplication

Another optimisation trick is related to multiplications and can already be found in the reduction functions. The standard way of multiplications as performed by the compiler is by using shifts and additions. This is very similar to the schoolbook method of multiplying 2 numbers in the decimal system. The 2 numbers, say abc and def are put underneath each other. Then, starting with the least significant digit, $fabc$ is calculated and put underneath def, such that the last digit is aligned with the f. Next, we shift one to the left, calculate $eabc$ and align the last digit with the e. Finally, we compute $dabc$ and align the rightmost digit with the d. We then proceed by computing the sum of the calculated partial products per column, carrying over the tens whenever the partial sum would exceed 9 and writing the remainder of the partial sum $\pmod{10}$. In the binary world, this works the same way. In fact, it is even easier because the partial products are only

			1	0	1	
		1	1	0	0	×
		1	1	0	0	
	0	0	0	0		
1	1	0	0			+
1	1	1	1	0	0	

Table 2: Example: binary multiplication of 5 (101) and 12 (1100).

formed by multiplications with 1 or 0. The way this is done, is simply by shifting bits. For example: we would like to multiply 5 (101 in binary) and 12 (1100 in binary). First we do 1 times 1100, then shift left and write 0 times 1100, then shift left and write 1 times 1100. We apply a binary `and` and we find 111100, which is indeed 60. This multiplication is illustrated in Table 2.

4.4.3 Compilation

Also worth mentioning is the way the files are compiled. In order to compare the C code and the MIPS64 assembly code properly, we make sure that we put the function in C in a separate file, with all the subfunctions and variables that it needs, and then compile using a certain flag. Then, we link the corresponding object file to a different C file (main.c) where we make a call to the function defined in the object file to do the actual measuring. We can also measure the assembly file, which has a .s extension, in a similar manner. The only difference is that it does not need to be compiled first [37].

In this chapter, we have seen the optimisation strategies that have been applied in order to reduce the amount of cycles of NHS, and the NTT in particular. The results can be found in Chapter 6. In the next chapter, we will make a small excursion to another computational problem on which we can base lattice-based cryptographic procols: the LWR problem. This is closely related to the LWE problem.

5 Adapting NHS to the LWR problem

In 2011 Banerjee et al. introduced a new lattice problem, that is closely related to the LWE problem: the LWR problem [4]. In this chapter, we will first explain the LWR problem. In Subsection 5.1.1 we describe how we can go from LWR to Ring-LWR. In Section 5.3 we describe the implementation of NHS scheme that is based on the RLWR problem. After that, we describe the failure probability of the "new scheme" in Section 5.4. In Section 5.5 we describe the possible passive attacks on the NHS scheme based on RLWR.

5.1 The LWR problem

The search version of the LWR problem is as follows.

Definition 5.1. LWR (search version).
Let the function $f_s : \mathbb{Z}_q^n \to \mathbb{Z}_p$ be given by

$$f_s(x) = \lfloor \langle x, s \rangle \rceil_p = \lfloor (p/q) \cdot \langle x, s \rangle \rceil, \tag{62}$$

where $s \in \mathbb{Z}_q^n$ is the secret key, $\langle x, s \rangle$ is the inner product of x and s mod q, and $\lfloor \cdot \rceil$ denotes rounding to the closest integer. Given independent samples of the form $(x, f_s(x))$, find the secret key s.

Essentially, instead of adding the noise to the result of the multiplication of the public matrix and the private vector in the LWE problem, every coefficient of the vector is mapped to its corresponding interval. A 1-dimensional example of 2 points being mapped is given in Figure 14. As Banerjee et al. mentioned in their paper, when both p and q, with $p < q$, are taken as powers of 2, the rounding can be done by dropping the last $\log_2(q/p)$ bits. Translating this interval number back to the original value is, similarly to the LWE problem, extremely hard for a good choice of p, because the inverse function maps the values in $\mathbb{Z}_p$ back to the centre of the intervals in $\mathbb{Z}_q$. The difference between that centre and the original input is similar to the addition of the noise in LWE, which depends on the parameter size of the probability distribution. Actually, in the LWR problem this "noise" comes from a uniformly random distribution, which is theoretically and practically even better than the LWE distribution. Theoretically because the failure probability should be a lot easier to compute, and practically because there is no need to sample the additional errors anymore. This saves a lot of randomness of the computer or device where the protocol is implemented.

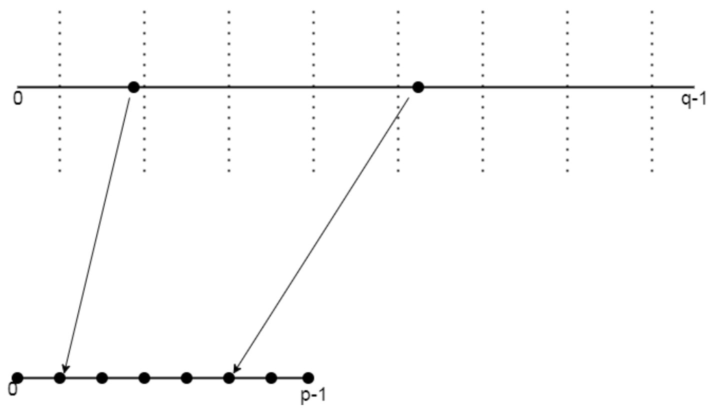


Figure 14: In the upper half we find 1-dimensional vectors, on a range that is divided into p intervals of length q/p. These vectors are mapped to their interval, which can be seen in the lower half.

There is also a decision version of the LWR problem.

Definition 5.2. LWR (decision version).
Given independent samples of pairs in $(\mathbb{Z}_q^n, \mathbb{Z}_p)$, distinguish pairs $(x, f_s(x))$ from uniformly randomly generated pairs.

5.1.1 Putting on the ring

The difference between regular LWR and LWR over a ring (RLWR) is analogous to the difference between LWE and Ring-LWE. Instead of sending an entire $n \times n$ matrix A, where each column is denoted by a_i, for $i = 0, \dots, n-1$, we simply use a single polynomial in the Ring-LWR problem, in which we represent the coefficients as a vector a. Working with the polynomial rings will have a major impact on the reduction of the message size and the corresponding speed. In the protocol, going from LWR to RLWR can be accomplished by changing the matrix A by a single vector a.

5.2 RLWR in practice

In this section, we give some explanation on the simplified NHS scheme based on RLWR, which is depicted in Figure 15. Note when we say vector, we mean the coefficient vector of a polynomial, and with vector multiplication we mean taking the negacyclic convolution of 2 polynomials.

Alice first generates her secret secret vector $s \leftarrow \psi_{16}$. She also generates a public vector $a \in \mathbb{Z}_q^n$. She computes the multiplication of these vectors, which will be uniformly spread in $\mathbb{Z}_q^n$. She then splits this range of possible values for each coefficient into p equal parts, which will all contain approximately q/p elements. Mapping these values from $\mathbb{Z}_q^n$ to $\mathbb{Z}_p^n$ can be done by multiplying with p/q and rounding the outcome to the closest integer.

There is, however, a possibility that the original value is positioned in the highest halve interval $[q-1-(q/2p), q-1)$, of which the rounding would result in p. That's the reason why she takes the output modulo p as her final result $b = \lfloor as \rceil_p \in \mathbb{Z}_p^n$. This will ensure that the b is uniformly random in the interval $[0, p-1]$. Alice sends her public key vector b together with the seed a to Bob, who, on his turn, generates a private vector $t \leftarrow \psi_{16}$ and calculates his public key $u \in \mathbb{Z}_p^n$ as $\lfloor at \rceil_p$. He then determines the shared secret ν, encodes it as k and includes the result in the calculation of $c = \lfloor bt \rceil_p + k$. Bob sends u and c to Alice, who wants to recover the shared secret ν. She can do this by making use of the fact that her public key is used in order to encrypt it. She subtracts $\lfloor us \rceil_p$ from c. In terms of formulae, this looks as follows:

$$\begin{aligned}
k' &= c - \lfloor us \rceil_p && (63)\\
&= \lfloor bt \rceil_p + k - \lfloor us \rceil_p && (64)\\
&= \lfloor \lfloor as \rceil_p t \rceil_p - \lfloor \lfloor at \rceil_p s \rceil_p + k && (65)\\
&= \lfloor ast \rceil_p - \lfloor ats \rceil_p + k && (66)\\
&\approx k. && (67)
\end{aligned}$$

Note that the operation $\lfloor \cdot \rceil_p$ is not associative. Because of the rounding errors that can occur, we can not simply subtract $\lfloor \lfloor at \rceil_p s \rceil_p$ from $\lfloor \lfloor as \rceil_p t \rceil_p$ to retrieve k exactly. Using the functions `NHSEncode` and `NHSDecode`, Alice and Bob can still (with high probability) agree on the same ν.

5.3 Implementation

Due to time restrictions, we have only made a simulation of the simplified NHS protocol based on the RLWR problem in Mathematica. This simulation `LWRsimple.nb` can be found in Appendix C. In order to make a good comparison to the original NHS protocol in terms of failure probability, we have chosen to use the same parameters n and q, and also the same probability distributions for the parameters.

Because initially the values did not meet our expectation, which we derive from the equations described in Section 5.4, we have decided to make another file `debug.nb`. In this file, the first function keeps simulating the protocol in Figure 15 and it stops whenever ν and ν' do not match entirely. Afterwards, there is a small function that outputs the position where the bit(s) is/are flipped, and it outputs all the variables with that index. The last, and maybe the most important function computes the same as the first function, without adding the shared secret. This way, we can see how big the difference between $\lfloor \lfloor as \rceil_p t \rceil_p$ and $\lfloor \lfloor at \rceil_p s \rceil_p$ can become, and how it is spread.

All the results of the files from the simulation can be found in Chapter 6.

5.4 Failure probability

As mentioned before, the difference between (R)LWE and (R)LWR is that the error in the former comes from a centered binomial distribution, whereas the error in the latter follows a discrete uniform distribution. This means that the error in the LWR variant can have values from $\{0, \ldots p-1\}$, all with equal probability $\frac{1}{p}$.

In order to see how big the difference between $\lfloor \lfloor as \rceil_p t \rceil_p$ and $\lfloor \lfloor at \rceil_p s \rceil_p$ is, we have to find our what values the rounding errors can take. In order to do so, we would like to express the function $\lfloor x \rceil_p$ in a formula:

$$\lfloor x \rceil_p = \lfloor x(p/q) \rceil \mod p. \tag{68}$$

As described in [1], for an $x \in \mathbb{R}$ the rounding function $\lfloor x \rceil$ is defined as $\lfloor x + \frac{1}{2} \rfloor \in \mathbb{Z}$. This implies that the rounded value can be translated back to the original by adding a term $\epsilon \in (-\frac{1}{2}, \frac{1}{2}]$. Using this, we see that the difference between the "real" value and the rounded value is

$$x(p/q) - \lfloor x \rceil_p = \epsilon + lp, \tag{69}$$

where ϵ still comes from $(-\frac{1}{2}, \frac{1}{2}]$, and lp comes from taking the result modulo p. Since $\lfloor x(p/q) \rceil$ can be at most p, it follows that l equals 0 or 1. $l = 1$ will only occur whenever $x(p/q) \geq p - \frac{1}{2}$, which happens with probability $\frac{p}{2q}$. We can rewrite the expression in (69) as

$$\lfloor x \rceil_p = x(p/q) - \epsilon - lp, \tag{70}$$

which we can use in the part of the protocol in Figure 15 where Alice computes the shared secret. This will give us a method to calculate the failure probability. Because Alice has to round the product of values which may have already been rounded, it is useful to indicate which error term comes from which rounding. This can be found in the simplified protocol in Figure 15. Following the notation from Figure 15, we can express all the rounded values

Alice		**Bob**
Generate $a \in \mathbb{Z}_q^n, s \leftarrow \psi_{16}^n$		Generate $t \leftarrow \psi_{16}^n$
(Priv. key) s		(Priv. key) t
(Pub. key) $b = \lfloor as \rceil_p$		(Pub. key) $u = \lfloor at \rceil_p$
(Rounding terms: ϵ, l)		(Rounding terms: ϵ', l')
	$\xrightarrow{a, b}$	
		Generate $\nu \in \{0,1\}^{256}$
		$k = \text{Encode}(\nu)$
		$c = \lfloor bt \rceil_p + k$
		(Rounding terms: ϵ'', l'')
	$\xleftarrow{u, c}$	
$k' = c - \lfloor us \rceil_p$		
(Rounding terms: ϵ''', l''')		
$= \lfloor bt \rceil_p + k - \lfloor us \rceil_p$		
$= \lfloor \lfloor as \rceil_p t \rceil_p + k - \lfloor \lfloor at \rceil_p s \rceil_p$		
$\approx k$		
$\nu' = \text{Decode}(k')$		

Figure 15: A simplified version of the *modified* NHS protocol, based on the RLWR problem instead of the RLWE problem. Encoding and decoding is done similarly to `NHSEncode` and `NHSDecode`, but with $\lfloor p/2 \rfloor$ instead of $\lfloor q/2 \rfloor$.

in terms of real values and the corresponding rounding errors. The result can be found in Equation (74).

$$\lfloor bt \rceil_p - \lfloor us \rceil_p = \lfloor \lfloor as \rceil_p t \rceil_p - \lfloor \lfloor at \rceil_p s \rceil_p \tag{71}$$

$$= bt\frac{p}{q} - \epsilon'' - l''p - us\frac{p}{q} + \epsilon''' + l'''p \tag{72}$$

$$= (as\frac{p}{q} - \epsilon - lp)t\frac{p}{q} - \epsilon'' - l''p - (at\frac{p}{q} - \epsilon' - l'p)s\frac{p}{q} + \epsilon''' + l'''p \tag{73}$$

$$= (-\epsilon - lp)t\frac{p}{q} - \epsilon'' - l''p - (-\epsilon' - l'p)s\frac{p}{q} + \epsilon''' + l'''p. \tag{74}$$

Suppose that we use pointwise multiplication of the polynomials. Then we see in Equation (73) that $ast\frac{p^2}{q^2}$ cancels out with $-ast\frac{p^2}{q^2}$, which is why we have no product of polynomials left in (74).

Combining this last equation with the upper bounds of the variables, we find an upper bound for the total difference as stated in Equation (71). In order to do so, we have to calculate the maximum values that s and t can take, with the corresponding probabilities. Looking at the distributions, we find that s and t can be at most 16 in absolute value (each with probability $2 \cdot 2^{-32}$). Since we take the difference modulo p, we get rid of $l''p$ and $l'''p$ in Equation (74). Also, we know that $|\epsilon| \leq 1/2$. Using this, we conclude that Equation (74) is bounded by

$$1 + (p/q)(t/2 + s/2) + (p^2/q)(t + s), \tag{75}$$

and since t and s are samples from the same distribution, we can rewrite this as

$$1 + (p/q)t + (p^2/q) \cdot 2t. \tag{76}$$

In order to decode the message correctly, this error must be smaller than $p/4$ (on average). This yield the following equation.

$$1 + (p/q)t + (p^2/q) \cdot 2t < p/4 \tag{77}$$

$$\frac{tp + 2tp^2}{p/4 - 1} < q. \tag{78}$$

From this, it follows that $p < 92$.

From the simulation in Appendix C, described in Section 5.3, it follows that the negacyclic convolution of the polynomials obviously has its impact on the calculations. Since this is beyond the scope of this thesis, the theoretical background where this negacyclic convolution is taken into account is not included.

5.5 Security analysis

Suppose that eavesdropper Eve intercepts the values a, b, u, and c. Suppose that she also knows the system parameters p and q. The question is if Eve can find any information about the secret vectors s and t, or about the shared secret ν.

We start with analysing b. Since b is defined as $\lfloor as \rceil_p$, Eve can multiply b with q/p to map the coefficients to the centre of the original intervals in $\mathbb{Z}_q^n$ again. Since the original values of as are spread uniformly over this interval, this problem is similar to the RLWE problem. Because of this translation from RLWR to RLWE, we can use the same security analysis as the one that has been used for the RLWE problem. If this interval in the RLWR version should have length 32, just like in the RLWE variant, we should choose p as $12,289/32 \approx 384$. Note however that the error in the RLWE problem comes from a binomial distribution that is centered around 0, meaning that noise will probably be small. In more than half of the cases, the error will be at maximum distance 3 from 0, and in more than 90% of the cases the maximum error will be at distance 5 from 0. Choosing intervals of length 10 means that the p has to be around $1,229$. Essentially, we can say that the smaller we choose our p, the bigger the intervals become, the more possibilities there are for values where the multiplication as comes from, and thus the more difficult it is to find the actual as.

5.6 Update August 4, 2017

The previously proposed scheme can be modified such that is resembles the old scheme better. Instead of working with the rounded values, it makes more sense to decompress the values first after receiving them. This decompression is done by multiplication with (q/p) and rounding to the nearest integer. We will illustrate this idea with a small example. Suppose that Alice computes her public key b by taking the product of a and s and applying the rounding function: $b = \lfloor as \rceil_p \in \mathbb{Z}_p^n$. She sends it to Bob, who decompresses it to $\tilde{b} \in \mathbb{Z}_q^n$. The difference between $\tilde{b}$ and as, an "earlier form" of b, is denoted by ϵ. Since the decompression maps every value back to the centre of the intervals, the error ϵ will be at most half such an interval. If we choose $p = 384$, the intervals would have length 32, which is the same as the error interval length in the RLWE version. The real difference between RLWE and RLWR is that the error in the former follows a binomial distribution, centered around 0, whereas the error in the latter is spread uniformly randomly over the interval.

Alice		**Bob**
Generate $a \in \mathbb{Z}_q^n, s \leftarrow \psi_{16}^n$		Generate $t \leftarrow \psi_{16}^n$
(Priv. key) s		(Priv. key) t
(Pub. key) $b = \lfloor as \rceil_p$		(Pub. key) $u = \lfloor at \rceil_p$
	$\xrightarrow{a, b}$	
		$\tilde{b} = (q/p) \cdot b$
		$(= as + \epsilon)$
		Generate $\nu \in \{0,1\}^{256}$
		$k = \text{Encode}(\nu)$
		$c = \lfloor \tilde{b}t + k \rceil_p$
	$\xleftarrow{u, c}$	
$\tilde{u} = (q/p) \cdot u$		
$(= at + \epsilon')$		
$\tilde{c} = (q/p) \cdot c$		
$(= \tilde{b}t + k + \epsilon'')$		
$k' = \tilde{c} - \tilde{u}s$		
$= \tilde{b}t + k + \epsilon'' - (at + \epsilon')s$		
$= (as + \epsilon)t + k + \epsilon'' - (at + \epsilon')s$		
$= ast + \epsilon t + k + \epsilon'' - ats - \epsilon' s$		
$= \epsilon t + k + \epsilon'' - \epsilon' s$		
$\approx k$		
$\nu' = \text{Decode}(k')$		

Figure 16: Modification of the RLWR version in Section 5.3. Encoding ν is done using function `NHSEncode` from NHS [2].

6 Results

In this chapter, we first describe the results from the MIPS64 assembly optimisation of the NTT compared to the original outcome. Afterwards, we will state the results of the simulation based on the RLWR problem. Every `.c`-file can be found in Appendix A.

6.1 NTT optimisation in MIPS64 assembly

In this section we present the performance of our code and compare it to other versions. An overview of the measurements of each NTT layer corresponding to every version of the code can be found in Table 3. The concept of layers in the NTT is explained in Section 4.3. The MIPS64 assembly code can be found in Appendix B and the 64-bit optimisations in C can be found in Appendix A. Before we state the results, we make a small remark on the way the results have been measured and compared.

In `ntt.c` in Appendix A, we give the original NH source code of the NTT function. Afterwards, still in Appendix A, we give our own general 64-bit optimisation of the NTT in `ntt_parallel.c`. In Appendix B, we give our own optimisations of the NTT, designed for MIPS64. These are spread over 4 `.s`-files: `pnttlevel0.s`, `pnttlevel1t4.s`, `pnttlevel5t8.s`, and `pnttlevel9.s`. In Appendix C, we state the Mathematica scripts that correspond to the RLWR implementation. In `LWRsimple.nb`, we give a simulation of the RLWR variant of the NHS protocol. In `debug.nb`, we give the script that can be run in order to find the magnitude of the errors and differences, such that we can say something about the failure probability.

6.1.1 Analysis of the NTT layers

Note that Table 3 has been created using a version of layer 9 in assembly that outputs doublewords of shape $0|c_i|0|c_{i+1}$, just as in Figure 12, instead of an array of halfwords c_i, as in Figure 11. Not converting the output to 16-bit integers will only have a minor influence on the eventual result. In order to retrieve the number of cycles, we have built a test file `test.c` in which we define 3 input arrays: `b`, which corresponds to the coefficient vector of the C code; `a`, a copy of `b` with $2 \cdot 1,024$ zeros concatenated at the end; and `omegas_montgomery`, which defines the ωs and thus is not changed. The reason for adding the zeros in `a` is that the assembly code of layer 0 first stores the results after performing the butterflies "behind" the first 1024 coefficients, such that no input value is changed.

Each time, the output of the assembly version is compared to the output of the C implementation, to verify that all values correspond. Furthermore, in Table 3 we count the cycles needed to run every single layer. Because the C code is compiled in a seperate file, with flag (-O0) or (-O3), we cannot measure a seperate layer (apart from layer 0). In order to determine the amount of cycles needed for one specific layer, say layer i, we simply count the cycles needed for layer 0 up to i and subtract the cycles that correspond to layer 0 up to $i-1$. We apply a similar method for measuring the assembly implementation for layers that belong to a block of multiple layers.

The code in `ntt.c` is the original code, as found in the NHS implementation. The code `ntt_parallel.c`, described in Paragraph 4.3.2.2, is the parallelised modification of `ntt.c`, being able to perform the butterfly operations on doublewords, each containing 2 coefficients instead of 1.

In Table 3 the performance of every NTT implementation is stated. Note that in `ntt_parallel.c` we run layer 0 in a non-parallel way. Afterwards, we convert our output to the correct format, i.e. put 16 bits of zero in between every coefficient. This makes sure that we have the correct doublewords that we can work with using the therefor designed functions. At the end of the loop, we have to remove the zeros again, such that the output is again an array of halfwords. These two conversions are included, together with the cycles needed to perform layer 0, between the brackets in layer 0.

Layer number	Assembly optimisation	`ntt.c` (-O0)	`ntt.c` (-O3)	`ntt_parallel.c` (-O0)	`ntt_parallel.c` (-O3)
0	**11,361**	**51,761**	**11,790**	**51,761 (79,935)**	**11,790 (16,831)**
1	8,366	69,164	17,937	39,198	10,763
2	5,536	51,790	12,097	28,466	10,000
3	7,072	69,266	17,985	39,258	12,318
4	5,536	51,994	12,262	28,586	10,026
1 - 4	**26,510**	**242,214**	**60,281**	**135,508**	**43,107**
5	8,082	69,674	18,150	39,498	12,375
6	5,376	52,810	12,934	29,066	10,115
7	6,912	71,306	18,813	40,458	12,623
8	5,376	56,074	15,622	30,986	10,440
5 - 8	**25,746**	**249,864**	**65,519**	**140,008**	**45,553**
9	**8,015**	**77,834**	**17,414**	**44,298**	**11,529**
Total	**71,632**	**621,673**	**155,004**	**399,749**	**117,020**

Table 3: The number of cycles needed to perform a layer per code.

6.1.2 Performance of the proposes NTT optimisations

As expected, the assembly implementation outperforms `ntt.c` easily. However, it also uses a lot fewer cycles than the parallelised C implementation, even with the somewhat agressive (-O3) flag. The reduction in the amount of cycles, expressed as a percentage, can be found in Table 4. The assembly

`ntt.c` (-O0)	`ntt.c` (-O3)	`ntt_parallel.c` (-O0)	`ntt_parallel.c` (-O3)
-88.5%	-53.8%	-82.1%	-38.8%

Table 4: The reduction in cycles using the optimised assembly code compared to every C implementation.

code is optimised for MIPS64, and not for other processors that with 64-bit registers. However, `ntt_parallel.c` is designed for any 64-bit processor. Using the compilation of this C implementation with the $-O3$ flag, we see that it takes 24.5% fewer cycles than the original `ntt.c`. Since the maximum size of a variable type in C is 64 bits, we can not parallelise the coefficients further. One could argue that it might also be possible to deal with 3 coefficients simultaneously, but this would only allow each coefficient to have $\lfloor \frac{64}{3} \rfloor = 21$ bits. Taking into account that the coefficients themselves are 16 bits long, this leaves us with 5 "overflow bits" per coefficient. Regarding the fact that we have to multiply the input coefficients with $q^{-1} = 12,287$ in the Montgomery reduction, which is too large to fit into 5 bits. This implies that there is no further parallelisation possible.

The lower amount of cycles in every even layer is due to the fact that the Barrett reduction is only performed in the odd layers, and not in the even ones.

6.1.3 Impact of the NTT optimisations across the NHS protocol

In order to see the impact that our optimisation has on the entire NHS protocol, we first state how much each function contributes to the protocol. This can be found in Table 5. We see that the FIPS202 functions also form a big part of the protocol.

	-O0	**-O1**	**-O2**	**-O3**
`SHA3-256`	19,671	7,572	9,945	5,519
`Parse(SHAKE-128(a))`	258,958	110,266	142,687	105,541
`multiplication`	112,669	37,920	37,920	37,159
`addition`	60,445	21,533	21,533	21,541
`encodeA`	110,388	34,542	34,544	26,431

Table 5: The number of cycles that are needed to perform the entire protocol, only one party, or only one function, using 4 different flags.

We also include the number of cycles needed to run the entire protocol, as well as the cycles needed for both parties separately. The results from these measurements using the original C code are stored in Table 6.

	-O0	**-O1**	**-O2**	**-O3**
NHS	8,575,978	2,977,592	2,877,620	2,559,095
Alice 1	2,901,161	1,011,736	986,295	871,441
Bob	4,539,638	1,568,441	1,516,147	1,345,723
Alice 2	1,227,007	397,503	375,210	342,235

Table 6: Number of cycles needed to perform (a part of) the NHS protocol, using `ntt.c`.

In these measurements, we have not used an optimised function yet. The results of the same measurements, using the parallelised C implementation can be found in Table 7.

	-O0	**-O1**	**-O2**	**-O3**
NHS	7,250,245	2,504,951	2,438,252	2,080,828
Alice 1	2,459,247	848,271	839,816	711,952
Bob	3,877,229	1,323,527	1,296,148	1,106,249
Alice 2	913,730	315,765	301,977	262,523

Table 7: Number of cycles needed to perform (a part of) the NHS protocol, using `ntt_parallel.c`.

As we can see, the number of cycles is already much lower than in Table 6. In Figure 5 in Section 2.4 it can be seen where the NTT function is called. Alice calls the NTT twice in the first part of the protocol. Bob then calls the function three times: twice as the normal NTT and once as the inverse NTT, which is esentially the NTT with different values for ω. In the final step, Alice also calls the inverse NTT function. The reduction of the cycles for the

NTT on these 6 places is what explains the reduction in the cycles in Table 7.

Last but not least, we state the results of the measurements of the MIPS64 assembly optimisation in Table 8.

	-O0	**-O1**	**-O2**	**-O3**
NHS	5,518,370	2,301,878	2,362,229	2,043,825
Alice 1	1,881,845	786,494	814,433	699,600
Bob	3,011,603	1,230,227	1,258,353	1,087,820
Alice 2	625,059	284,880	289,303	256,327

Table 8: Number of cycles needed to perform (a part of) the NHS protocol, using the MIPS64 assembly optimisation.

Again, these results are improvements compared to Table 6, but also compared to Table 7. The latter can be explained by the fact that we have designed the MIPS64 assembly optimisation especially for the MIPS64 architecture, and so we have been able to really exploit every of its properties.

In Table 9 we give an overview of the number of cycles needed to perform the entire NHS protocol for all the implementations for 4 different flags.

	-O0	**-O1**	**-O2**	**-O3**
NHS with `ntt.c`	8,575,978	2,977,592	2,877,620	2,559,095
NHS with `ntt_parallel.c`	7,250,245	2,504,951	2,438,252	2,080,828
NHS with MIPS64 assembly*	5,518,370	2,301,878	2,362,229	2,043,825

Table 9: Measurements of the entire NHS protocol.

Moreover, we depict the performance first using the original `ntt.c`, then with `ntt_parallel.c` instead of `ntt.c`, and finally with the MIPS64 assembly implementation*. Each result is the average of 100 measurements. *Since this code is already optimised, there is no reason to do it again. We can compile the rest of the C files, such as FIPS202 (which contains the Keccak hash functions SHA3-256 and SHAKE-128[52], described in Subsection 2.4.2) and ChaCha20 (used in the generation of the centered binomial samples as described in Subsection 2.4.1), that are used in NHS with the flags though, in order to get a fair comparison.

6.2 RLWR simulation

In the initial simulation of the NHS protocol based on the RLWR problem, which can be found in `LWRsimple.c` in Appendix C, we run the protocol N times, with parameters $n = 1,024$ and $q = 12,289$. In Table 10 we state the amount of times that the bitstrings ν and ν' contain at least one mismatched bit. Note that the number of flipped bits, if it is bigger than 0, does not matter, since the outputs ν and ν' are hashed to get to the shared key. We denote the number of mismatches of ν and ν' by "# errors".

p	# errors
8	0
16	2
32	98
64	499

Table 10: Number of mismatches of ν and ν' for different choices of p, using $N = 500$.

The results of the simulation when the number of runs of the protocol is incremented to $1,000$, are written in Table 11.

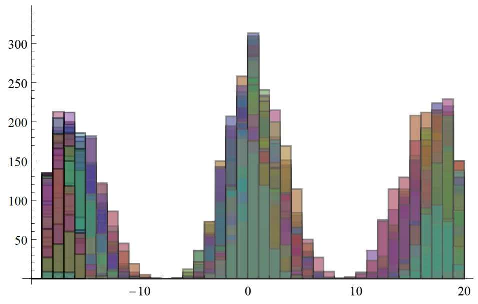
p	# errors
8	0
16	3

Table 11: Number of mismatches ν and ν' for different choices of p, using $N = 1,000$.

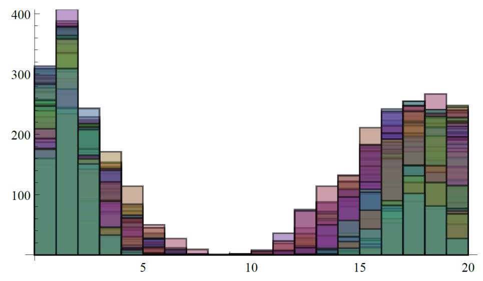
Using the file `debug.nb` in Appendix C, we can simulate a part of the version of the NHS protocol based on the RLWR problem. We simply leave out the addition of the shared secret, such that we can simulate the difference between $\lfloor\lfloor as\rceil_p t\rceil_p$ and $\lfloor\lfloor at\rceil_p s\rceil_p$. Again, we use $n = 1,024$ and $q = 12,289$. The number of times that the protocol is run is 1000 every time. We experiment with different values of p. For each value of p, we plot the difference $\lfloor\lfloor as\rceil_p t\rceil_p - \lfloor\lfloor at\rceil_p s\rceil_p$, denoted by $c - v$, and the absolute difference $|\lfloor\lfloor as\rceil_p t\rceil_p - \lfloor\lfloor at\rceil_p s\rceil_p|$, denoted by $|c - v|$.

In Figure 17 we see the results of the simulation using $p = 20$.

These results contain a lot of values that are larger than or equal to $p/4 = 20$. We try again with a smaller value of p, namely $p = 16$. The corresponding



(a) The differences $c - v$.

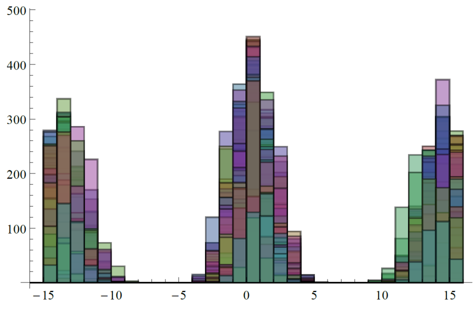


(b) The differences $|c - v|$.

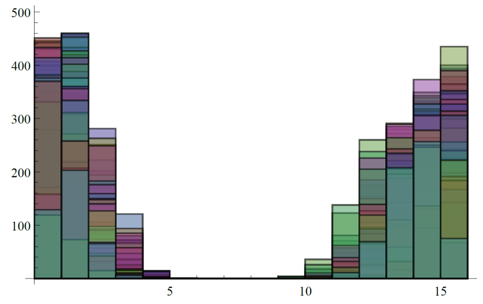
Figure 17: The results of the simulation using $p = 20$.

results can be found in Table 18.

Still, the errors can become too large. In Table 19, we state the results



(a) The differences $c - v$.

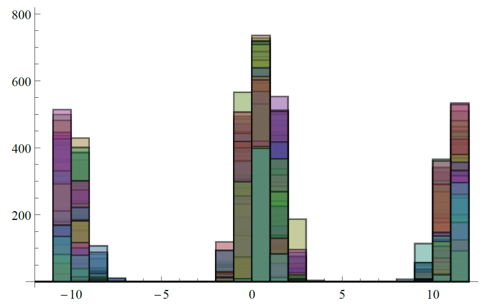


(b) The differences $|c - v|$.

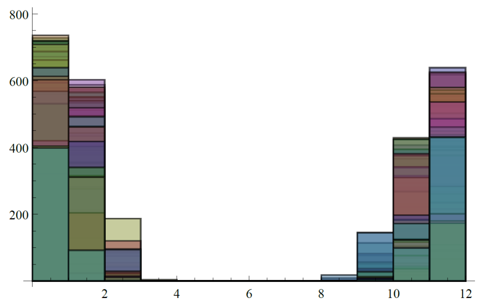
Figure 18: The results of the simulation using $p = 16$.

for $p = 12$.

Still there are cases where the errors get larger than $p/4$. Therefore, we simulate using $p = 8$ and give the results in Table 20.

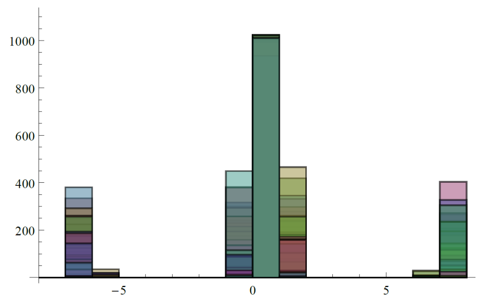


(a) The differences $c - v$.

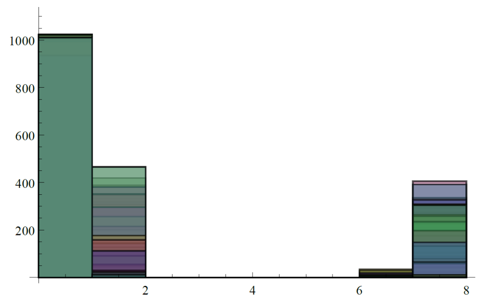


(b) The differences $|c - v|$.

Figure 19: The results of the simulation using $p = 12$.



(a) The differences $c - v$.

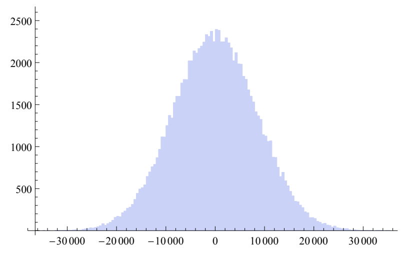


(b) The differences $|c - v|$.

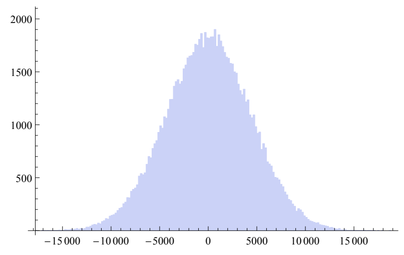
Figure 20: The results of the simulation using $p = 8$.

6.3 Update August 4, 2017

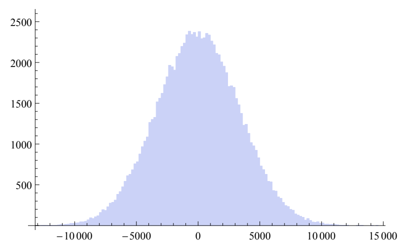
In `debugUpdate.nb`, in Appendix C, we have adapted `debug.nb` to the latest version of the protocol, described in Section 5.6. We print how the coefficients of ϵs are distributed in order to compare it to error terms in NHS. In `debugUpdate.nb` we construct 2 polynomials with $n = 100,000$ coefficients. The coefficients from polynomial ϵ are distributed uniformly random over the interval $[-\lfloor q/2p \rfloor, \lfloor q/2p \rfloor]$, and $s \leftarrow \psi_{16}^{n}$. We calculate the negacyclic convolution of the polynomials and print a histogram of the coefficients. In order to see the impact that the interval length has on the distribution of the coefficients, we use different values of it in every run. The interval length is calculated as $\lfloor q/p \rfloor$. Each of the plots given below looks like a Gaussian distribution. That's why, for every plot, we state the mean μ and the standard deviation σ, together with the smallest coefficient "min" and the largest coefficient "max". In Figure 21, we show the histograms of the coefficients for 7 choices of interval length.



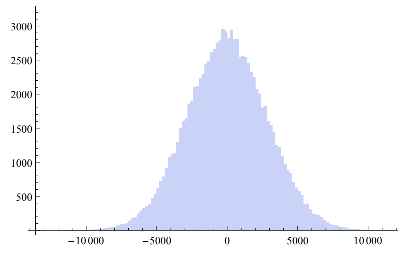
(a) Interval length 32.



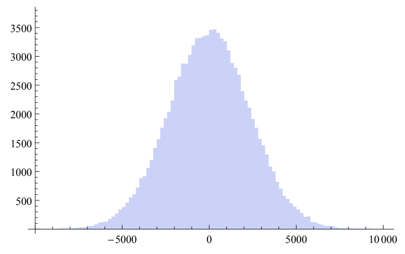
(b) Interval length 16.



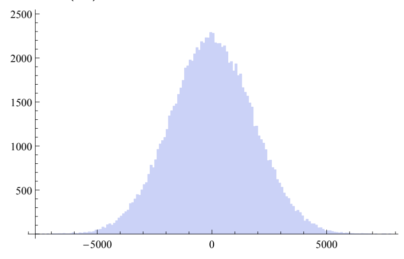
(c) Interval length 12.



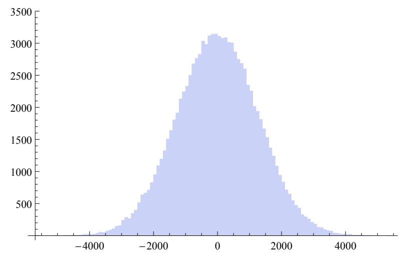
(d) Interval length 10.



(e) Interval length 8.



(f) Interval length 6.



(g) Interval length 4.

Figure 21: Illustration of the distribution of the coefficients of ϵs for each interval length. The corresponding variables μ, σ, min, and max can be found in Table 12.

In Table 12 we state the variables that correspond to the distributions in Figure 21. In order to compare the new protocol to NHS, based on RLWE,

Interval length	32	16	12	10	8	6	4
μ	-18.96	30.93	-6.79	-6.81	3.48	3.23	-0.01
σ	8,502.09	4,381.07	3,366.12	2,819.44	2,329.68	1,794.01	1,263.42
min	-36,261	-17,687	-13,115	-13,523	-9,965	-7,648	-5,610
max	35,259	18,727	14,413	11,566	10,118	7,711	5,394

Table 12: The variables that correspond to the distributions shown in Figure 21.

we also make a simulation in which both polynomials s and ϵ come from the distribution ψ_{16}^{n}. We use $n = 100,000$, calculate the negacyclic convolution of s and ϵ, and show in Figure 22 how the coefficients are distributed. Also, we state the corresponding variables. If we compare this to the results of the

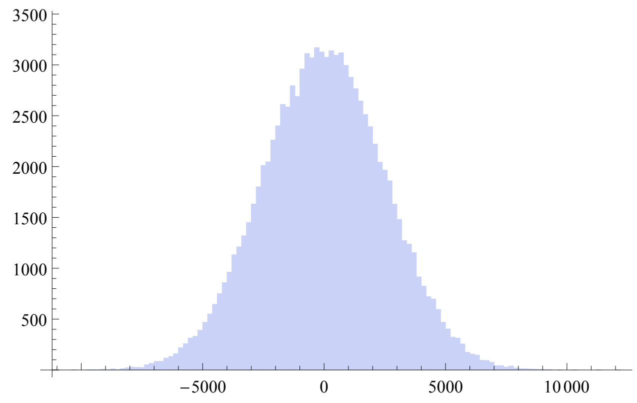


Figure 22: The distribution of the coefficients of ϵs, with mean $\mu = -3.81$, the standard deviation $\sigma = 2,517.64$, min $= -11,154$, and max $= 12,092$.

simulation in `debugUpdate.nb`, stated in Table 12, we see that the RLWE variant behaves comparably to the RLWR variant with interval lengths 10 and 8. Assuming that all the distributions are approximately Gaussian, or sub-gaussian, choosing an interval length of 8 or smaller in the RLWR variant will give smaller errors than the RLWE variant. Afterwards, we can follow the same reasoning as Appendix D from NH [1] to get the same failure probability as in NH(S), or an even smaller one.

Contrary to choosing a large interval, the choice of a small interval has the disadvantage that the messages are relatively large. Suppose we want to work with intervals of length 8, then p consists of $\lfloor \log(12,289/8)/\log(2) \rfloor + 1 = 11$ bits. One has to make a trade-off between failure probability and message size. Furthermore, the RLWR protocol makes use of divisions by q and p. Our current choice of q is very unfortunate. Division by $q = 12,289$ is a very costly operation in practice. If q and p are chosen as powers of 2, however, the division can simply be done by shifting the bitstring of q with $\log(p)$ positions to the right.

7 Discussion

Looking at Table 3 in Chapter 6 it can be concluded that the MIPS64 assembly optimisation of the NTT needs less than half the amount of cycles of the reference implementation `ntt.c`, compiled with the (-O3) flag. This is also in the order of magnitude that we had expected. Because we are able to deal with twice the number of coefficients at the same time in 9 out of 10 layers, it makes sense that we only need approximately half the amount of cycles. Since we cannot store more than 2 coefficients in a register, this MIPS64 assembly optimisation should also be close to optimal. Maybe there are tricks to reduce the amount of cycles even more, e.g. by putting multiple constants into 1 register and applying a bitmask every time they are called, but this will only give a relatively small improvement. Something that can be worth looking at is shifting the output string in the last layer to the first positions, which will automatically come at the cost of more instructions.

Regarding the RLWR problem it follows from the simulation that the negacyclic convolutions of the polynomials have an impact on the calculations. This needs more time to be investigated theoretically. From the simulation, it seems that $p = 8$ is a safe value to pick, but it would be nice to justify this.

7.1 Update (August 4, 2017)

From the simulation it follows that picking p large enough in the RLWR variant of the protocol, we can get the same (or an even lower) failure probability as the one in NH(S). However, this is only based on the simulations and the assumptions that the distributions are (approximately) Gaussian. Theoretically, it would be better if we could also justify this by working out the joint probability distributions. However, this turned out to be beyond the scope of this project.

8 Future work

Due to the limited time that I have been able to work on this project, I have had to make certain subjects to investigate. There are still some topics left that could be interesting to look at. One of these topics is the optimisation of another important and frequently used mechanism in the NewHope-Simple protocol: the SHA3 function.

Also, it can be interesting to not only look at the simulation of the RLWR problem, but also the implementation in C, in a similar way as the source code of NHS. In order to do so, we first would have to do research at the choice of q and p. As mentioned in the discussion, these should both be powers of 2 to make the divisions and multiplications not too expensive.

Another topic that would be more interesting in terms of theory (and eventually maybe also in practice) is an NHS-like protocol that is based on a joint version of the RLWE and RLWR problem. This is supposedly even harder than the RLWE and RLWR problem, but it also probably also more cycle-consuming.

9 Conclusion

The aim of this thesis is to optimise the post-quantum key-exchange scheme NHS. We have done this both practically and theoretically. Practically by optimising one of the building blocks of the NHS scheme, the NTT, for 64-bit processors in general and the MIPS64 architecture in particular. The practical improvements have been done by modifying the NHS protocol, such that the underlying lattice problem is the RLWR problem instead of the RLWE problem.

The MIPS64 assembly optimisation of the NTT is provably faster than the original reference code, so we would strongly recommend to use this optimisation on MIPS64 processors. In the entire protocol, it will save more than 20% of the cycles compared to the original implementation compiled with (-O3). Also, if one has access to a 64-bit processor that does not have the MIPS architecture, we recommend using the parallelised version of the NTT in C, which saves more than 18.5% of the cycles on the protocol compared to the original reference code.

Theoretically, the RLWR variant of the NHS protocol is also recommended, because of the fact that it requires fewer random bits from the device where it is run. Also, the RLWR turns out to be very similar to (and maybe even better than) RLWE. From a practical point of view it is would also be recommended, since the messages that have to be sent are automatically compressed, because of the rounding function. However, compression size also has its impact on the failure probability. The trade-off between size and failure probability is up to the implementer. All in all, it is an interesting alternative for the RLWE version of NH(S).

References

[1] Erdem Alkim, Léo Ducas, Thomas Pöppelmann, and Peter Schwabe. Post-quantum key exchange – a new hope. In *Proceedings of the 25th USENIX Security Symposium*. USENIX Association, 2016. Retrieved from `http://eprint.iacr.org/2015/1092`.

[2] Erdem Alkim, Léo Ducas, Thomas Pöppelmann, and Peter Schwabe. NewHope without reconciliation. IACR Cryptology ePrint Archive report 2016/1157. Retrieved from `https://eprint.iacr.org/2016/1157`.

[3] Oded Regev. On lattices, learning with errors, random linear codes, and cryptography. In *Proc. 37th ACM Symp. on Theory of Computing (STOC)*, pages 84–93, 2005.

[4] Abhishek Banerjee, Chris Peikert, and Alon Rosen. Pseudorandom Functions and Lattices. IACR Cryptology ePrint Archive report 2011/401. Retrieved from `https://eprint.iacr.org/2011/401.pdf`.

[5] Imagination Technologies LTD. and/or its Affiliated Group Companies. MIPS®Architecture For Programmers Volume I-A: Introduction to the MIPS64®Architecture. Retrieved from `http://cdn2.imgtec.com/documentation/MD00083-2B-MIPS64INT-AFP-06.01.pdf`.

[6] Department of Computer Science, Northern Illinois University. RISC - Reduced Instruction Set Computer. Retrieved from `http://faculty.cs.niu.edu/~berezin/463/lec/05/`.

[7] Imagination Technologies LTD. and/or its Affiliated Group Companies. MIPS®Architecture For Programmers Volume II-A: The MIPS64®Instruction Set Reference Manual. Revision 6.05, June 3, 2016. Retrieved from `https://imagination-technologies-cloudfront-assets.s3.amazonaws.com/documentation/MIPS_Architecture_MIPS64_InstructionSet_%20AFP_P_MD00087_06.05.pdf`.

[8] Sukumar Ghosh. MIPS registers. Retrieved from `http://homepage.divms.uiowa.edu/~ghosh/1-28-10.pdf`.

[9] MIPS Technologies Inc. Using the MIPS32®34K®Core Performance Counters. Revision 01.01, August 22, 2011. Retrieved from `http://cdn.imgtec.com/mips-documentation/login-required/using_the_mips32_34k_core_performance_counters.pdf`.

[10] Fred Cohen. A Short History of Cryptography. Retrieved from http://all.net/edu/curr/ip/Chap2-1.html.

[11] R.L. Rivest, A. Shamir, and L. Adleman. A Method for Obtaining Digital Signatures and Public-Key Cryptosystems. Retrieved from http://people.csail.mit.edu/rivest/Rsapaper.pdf.

[12] Peter Shor. Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer. Retrieved from https://arxiv.org/pdf/quant-ph/9508027.pdf.

[13] Davide Castelvecchi. Quantum computers ready to leap out of the lab in 2017. In *Nature 541*. Retrieved from http://www.nature.com/news/quantum-computers-ready-to-leap-out-of-the-lab-in-2017-1.21239.

[14] Bernstein et. al. Post-Quantum Cryptography. Retrieved from https://www.researchgate.net/profile/Nicolas_Sendrier/publication/226115302_Code-Based_Cryptography/links/540d62d50cf2df04e7549388/Code-Based-Cryptography.pdf.

[15] Vadim Lyubashevsky, Chris Peikert, and Oded Regev. On Ideal Lattices and Learning with Errors Over Rings. IACR Cryptology ePrint Archive report 2012/230. https://eprint.iacr.org/2012/230.pdf.

[16] David Deutsch. Quantum computation. A comprehensive and inspiring guide to quantum computing. In *Physics World*, 1/6/92. Retrieved from https://www.doc.ic.ac.uk/~nd/surprise_97/journal/vol4/spb3/#1.1%20Quantum%20computer%20basics.

[17] Ronald de Wolf. Quantum Computing: Lecture Notes. Retrieved from http://homepages.cwi.nl/~rdewolf/qcnotes.pdf.

[18] Philipp Jakubeit. NewHope for ARM: An Efficient Implementation of the Post-Quantum Ephemeral Key Exchange NewHope for the ARMv6-M Architecture. Retrieved from www.ru.nl/publish/pages/769526/philipp_jakubeit.pdf.

[19] Chris Peikert. A Brief History of Lattices in Cryptography. Retrieved from https://web.eecs.umich.edu/~cpeikert/lic13/lec01.pdf.

[20] Chris Peikert. SVP, Gram-Schmidt, LLL. Retrieved from https://web.eecs.umich.edu/~cpeikert/lic13/lec02.pdf.

[21] Oded Regev. The Learning with Errors Problem. Retrieved from `http://www.cims.nyu.edu/~regev/papers/lwesurvey.pdf`.

[22] Joppe W. Bos, Craig Costello, Michael Naehrig, and Douglas Stebila. Post-quantum key exchange for the TLS protocol from the ring learning with errors problem. Retrieved from `https://eprint.iacr.org/2014/599.pdf`.

[23] Daniel Bernstein. ChaCha, a variant of Salsa20. Retrieved from `https://cr.yp.to/chacha/chacha-20080128.pdf`.

[24] M.S. Schmalz. Organization of Computer Systems: Pipelining. Retrieved from `https://www.cise.ufl.edu/~mssz/CompOrg/CDA-pipe.html`.

[25] Peter Montgomery. Modular Multiplication Without Trial Division. In *Mathematics of Computation, Vol. 44 No. 170. (Apr., 1985), pp. 519-521.* Retrieved from `https://cseweb.ucsd.edu/classes/fa06/cse246/montgomery.pdf`.

[26] Mark Buxton. Haswell New Instruction Descriptions Now Available! Retrieved from `https://software.intel.com/en-us/blogs/2011/06/13/haswell-new-instruction-descriptions-now-available`.

[27] Tim Güneysu, Tobias Oder, Thomas Pöppelmann, and Peter Schwabe. Software Speed Records for Lattice-Based Signatures. In *Post-Quantum Cryptography (2013), P. Gaborit, Ed., vol. 7932 of LNCS, Springer, pp. 67–82.* Retrieved from `https://cryptojedi.org/papers/lattisigns-20130328.pdf`.

[28] Erdem Alkim, Philipp Jakubeit, and Peter Schwabe. A new hope on ARM Cortex-M. Retrieved from `https://cryptojedi.org/papers/newhopearm-20160803.pdf`.

[29] Arm. Cortex-M Series Family. Retrieved from `http://www.arm.com/products/processors/cortex-m`.

[30] Arm. Cortex-M4. Retrieved from `https://developer.arm.com/products/processors/cortex-m/cortex-m4`.

[31] Paul Barrett. Implementing the Rivest Shamir and Adleman Public Key Encryption Algorithm on a Standard Digital Signal Processor. In *Proc. CRYPTO'86, pages 311–323, 1986.*

[32] David Maier. Polynomials and the Fast Fourier Transform (FFT) [PowerPoint slides]. Retrieved from `http://web.cecs.pdx.edu/~maier/cs584/Lectures/lect07b-11-MG.pdf`.

[33] Richard Crandall and Carl Pomerance. Prime numbers - a computational perspective. `http://thales.doa.fmph.uniba.sk/macaj/skola/teoriapoli/primes.pdf`.

[34] Eric W. Weisstein. Primitive Root of Unity. In *MathWorld–A Wolfram Web Resource.* `http://mathworld.wolfram.com/PrimitiveRootofUnity.html`.

[35] Tim Weenink. Transformations and their applications in cryptology. This paper is the result of the course Capita Selecta (Research Topic 1) of the Technical University of Eindhoven. (2017) *Send an e-mail if you'd like to see the document.*

[36] James Cooley and John Tukey. An algorithm for the machine calculation of complex Fourier series. In *Math. Comput. 19: 297–301. (1965).*

[37] GCC. Using the GNU Compiler Collection (GCC): Optimize Options. Retrieved from `https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html`.

[38] Bracewell, R. Convolution Theorem. In *The Fourier Transform and Its Applications, 3rd ed. New York: McGraw-Hill, pp. 108-112, 1999.*

[39] Eleanor Chu and Alan George. Inside the FFT black box. Serial and Parallel Fast Fourier Transform Algorithms. In *Computational mathematical series.* Retrieved from `http://dsp-book.narod.ru/FFTBB/0270_PDF_C04.pdf`.

[40] Craig Gentry. Fully Homomorphic Encryption Using Ideal Lattices. In *the 41st ACM Symposium on Theory of Computing (STOC), 2009.* Retrieved from `https://www.cs.cmu.edu/~odonnell/hits09/gentry-homomorphic-encryption.pdf`.

[41] Peter van Emde Boas. Another NP-complete problem and the complexity of computing short vectors in a lattice. In *Technical Report 8104. University of Amsterdam, Department of Mathematics, Netherlands, (1981).*

[42] Miklós Ajtai. The shortest vector problem in L2 is NP-hard for randomized reductions. In *Proceedings of the thirtieth annual ACM symposium on Theory of computing. Dallas, Texas, United States: ACM. pp. 10–19.* Retrieved from http://www.csie.nuk.edu.tw/~cychen/Lattices/Generating%20Hard%20Instances%20of%20Lattice%20Problems.pdf.

[43] Arjen Lenstra, Hendrik Lenstra, and László Lovász. Factoring polynomials with rational coefficients. In *Mathematische Annalen. 261 (4): 515–534.*

[44] Daniele Micciancio. The shortest vector in a lattice is hard to approximate to within some constant. In *SIAM Journal on Computing 30.6 (2001), pp. 2008–2035.* Retrieved from https://cseweb.ucsd.edu/~daniele/papers/SVP.pdf.

[45] O. Goldreich, D. Micciancio, S. Safra, and J.-P. Seifert. Approximating shortest lattice vectors is not harder than approximating closest lattice vectors. In *Information Processing Letters 71.2 (1999), pp. 55–61.* Retrieved from https://cseweb.ucsd.edu/~daniele/papers/GMSS.pdf.

[46] U. Feige and D. Micciancio. The inapproximability of lattice and coding problems with preprocessing. In *Journal of Computer and System Sciences 69.1 (2004), pp. 45–67.* Retrieved from https://cseweb.ucsd.edu/~daniele/papers/GapCVPP.pdf.

[47] Guillaume Hanrot and Damien Stehlé. Worst-Case Hermite-Korkine-Zolotarev Reduced Lattice Bases. In *[Research Report] RR-6422, INRIA. 2008, pp.25.* Retrieved from https://core.ac.uk/download/pdf/52327979.pdf.

[48] Phong Nguyen. Worst-case to Average-case Reductions For Lattice Problems. [Powerpoint slides]. Retrieved from http://www.lorentzcenter.nl/lc/web/2016/834/presentations/Nguyen.

[49] Jintai Ding, Xiang Xie, Xiaodong Lin. A Simple Provably Secure Key Exchange Scheme Based on the Learning with Errors Problem. In *IACR Cryptology ePrint Archive 2012.1 (2012), p. 688.* Retrieved from https://eprint.iacr.org/2012/688.pdf.

[50] Chris Peikert. Public-key cryptosystems from the worst-case shortest vector problem. In *Proceedings of the forty-first annual ACM symposium on Theory of computing. 2009, pp. 333–342.* Retrieved

from http://drops.dagstuhl.de/opus/volltexte/2009/1892/pdf/08491.PeikertChris.Paper.1892.pdf.

[51] Chris Peikert. Lattice cryptography for the Internet. In *Post-Quantum Cryptography. 2014, pp. 197–219.* Retrieved from https://web.eecs.umich.edu/~cpeikert/pubs/suite.pdf.

[52] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. The Keccak sponge function family. Retrieved from http://keccak.noekeon.org/.

[53] Computer Security Division and Computer Security Resource Center. FIPS PUBLICATIONS. Retrieved from http://csrc.nist.gov/publications/PubsFIPS.html.

[54] Eli Biham. A Fast New DES Implementation in Software. Retrieved from http://www.cs.technion.ac.il/users/wwwb/cgi-bin/tr-get.cgi/1997/CS/CS0891.pdf.

[55] Daniele Micciancio. Closest Vector Problem. In *Encyclopedia of Cryptography and Security, pp 212-214.*

[56] Imagination Technologies. MIPS DSP. Retrieved from https://www.imgtec.com/mips/architectures/dsp/.

[57] Imagination Technologies. MIPS SIMD. Retrieved from https://www.imgtec.com/mips/architectures/simd/.

[58] Alex Voica. Five most iconic devices to use MIPS CPUs. Retrieved from https://www.imgtec.com/blog/five-most-iconic-devices-to-use-mips-cpus/.

Appendices

A NHS 64-bit optimisation (in C)

First we describe the original file `ntt.c` as it is found in the reference [2].

```
#include "inttypes.h"
#include "ntt.h"
#include "params.h"
#include "reduce.h"

static uint16_t bitrev_table[PARAM_N] = {
  0,512,256,768,128,640,384,896,64,576,320,832,192,704,448,960,32,544,
  288,800,160,672,416,928,96,608,352,864,224,736,480,992,
  16,528,272,784,144,656,400,912,80,592,336,848,208,720,464,976,48,560,
  304,816,176,688,432,944,112,624,368,880,240,752,496,1008,
  8,520,264,776,136,648,392,904,72,584,328,840,200,712,456,968,40,552,
  296,808,168,680,424,936,104,616,360,872,232,744,488,1000,
  24,536,280,792,152,664,408,920,88,600,344,856,216,728,472,984,56,568,
  312,824,184,696,440,952,120,632,376,888,248,760,504,1016,
  4,516,260,772,132,644,388,900,68,580,324,836,196,708,452,964,36,548,
  292,804,164,676,420,932,100,612,356,868,228,740,484,996,
  20,532,276,788,148,660,404,916,84,596,340,852,212,724,468,980,52,564,
  308,820,180,692,436,948,116,628,372,884,244,756,500,1012,
  12,524,268,780,140,652,396,908,76,588,332,844,204,716,460,972,44,556,
  300,812,172,684,428,940,108,620,364,876,236,748,492,1004,
  28,540,284,796,156,668,412,924,92,604,348,860,220,732,476,988,60,572,
  316,828,188,700,444,956,124,636,380,892,252,764,508,1020,
  2,514,258,770,130,642,386,898,66,578,322,834,194,706,450,962,34,546,
  290,802,162,674,418,930,98,610,354,866,226,738,482,994,
  18,530,274,786,146,658,402,914,82,594,338,850,210,722,466,978,50,562,
  306,818,178,690,434,946,114,626,370,882,242,754,498,1010,
  10,522,266,778,138,650,394,906,74,586,330,842,202,714,458,970,42,554,
  298,810,170,682,426,938,106,618,362,874,234,746,490,1002,
  26,538,282,794,154,666,410,922,90,602,346,858,218,730,474,986,58,570,
  314,826,186,698,442,954,122,634,378,890,250,762,506,1018,
  6,518,262,774,134,646,390,902,70,582,326,838,198,710,454,966,38,550,
  294,806,166,678,422,934,102,614,358,870,230,742,486,998,
  22,534,278,790,150,662,406,918,86,598,342,854,214,726,470,982,54,566,
  310,822,182,694,438,950,118,630,374,886,246,758,502,1014,
  14,526,270,782,142,654,398,910,78,590,334,846,206,718,462,974,46,558,
  302,814,174,686,430,942,110,622,366,878,238,750,494,1006,
  30,542,286,798,158,670,414,926,94,606,350,862,222,734,478,990,62,574,
  318,830,190,702,446,958,126,638,382,894,254,766,510,1022,
  1,513,257,769,129,641,385,897,65,577,321,833,193,705,449,961,33,545,
  289,801,161,673,417,929,97,609,353,865,225,737,481,993,
  17,529,273,785,145,657,401,913,81,593,337,849,209,721,465,977,49,561,
  305,817,177,689,433,945,113,625,369,881,241,753,497,1009,
  9,521,265,777,137,649,393,905,73,585,329,841,201,713,457,969,41,553,
  297,809,169,681,425,937,105,617,361,873,233,745,489,1001,
  25,537,281,793,153,665,409,921,89,601,345,857,217,729,473,985,57,569,
  313,825,185,697,441,953,121,633,377,889,249,761,505,1017,
  5,517,261,773,133,645,389,901,69,581,325,837,197,709,453,965,37,549,
  293,805,165,677,421,933,101,613,357,869,229,741,485,997,
  21,533,277,789,149,661,405,917,85,597,341,853,213,725,469,981,53,565,
  309,821,181,693,437,949,117,629,373,885,245,757,501,1013,
  13,525,269,781,141,653,397,909,77,589,333,845,205,717,461,973,45,557,
  301,813,173,685,429,941,109,621,365,877,237,749,493,1005,
  29,541,285,797,157,669,413,925,93,605,349,861,221,733,477,989,61,573,
  317,829,189,701,445,957,125,637,381,893,253,765,509,1021,
  3,515,259,771,131,643,387,899,67,579,323,835,195,707,451,963,35,547,
  291,803,163,675,419,931,99,611,355,867,227,739,483,995,
  19,531,275,787,147,659,403,915,83,595,339,851,211,723,467,979,51,563,
  307,819,179,691,435,947,115,627,371,883,243,755,499,1011,
  11,523,267,779,139,651,395,907,75,587,331,843,203,715,459,971,43,555,
  299,811,171,683,427,939,107,619,363,875,235,747,491,1003,
  27,539,283,795,155,667,411,923,91,603,347,859,219,731,475,987,59,571,
  315,827,187,699,443,955,123,635,379,891,251,763,507,1019,
  7,519,263,775,135,647,391,903,71,583,327,839,199,711,455,967,39,551,
  295,807,167,679,423,935,103,615,359,871,231,743,487,999,
  23,535,279,791,151,663,407,919,87,599,343,855,215,727,471,983,55,567,
  311,823,183,695,439,951,119,631,375,887,247,759,503,1015,
  15,527,271,783,143,655,399,911,79,591,335,847,207,719,463,975,47,559,
  303,815,175,687,431,943,111,623,367,879,239,751,495,1007,
  31,543,287,799,159,671,415,927,95,607,351,863,223,735,479,991,63,575,
  319,831,191,703,447,959,127,639,383,895,255,767,511,1023
};

void bitrev_vector(uint16_t* poly)
{
    unsigned int i,r;
    uint16_t tmp;
    for(i = 0; i < PARAM_N; i++)
    {
        r = bitrev_table[i];
        if (i < r)
        {
          tmp = poly[i];
          poly[i] = poly[r];
          poly[r] = tmp;
        }
    }
}

void mul_coefficients(uint16_t* poly, const uint16_t* factors)
{
    unsigned int i;

    for(i = 0; i < PARAM_N; i++)
      poly[i] = montgomery_reduce((poly[i] * factors[i]));
}



/* GS_bo_to_no; omegas need to be in Montgomery domain */
void ntt(uint16_t * a, const uint16_t* omega)
{
  int i, start, j, jTwiddle, distance;
  uint16_t temp, W, temp0;


  for(i=0;i<10;i+=2)
  {
    // Even level
    distance = (1<<i);
    for(start = 0; start < distance;start++)
    {
      jTwiddle = 0;
      for(j=start;j<PARAM_N-1;j+=2*distance)
      {
        W = omega[jTwiddle++];
        temp = a[j];
        a[j] = (temp + a[j + distance]); // Omit reduction (be lazy)
        a[j + distance] = montgomery_reduce((W * ((uint32_t)temp
+ 3*PARAM_Q - a[j + distance])));
      }
    }

    // Odd level
    distance <<= 1;
    for(start = 0; start < distance;start++)
    {
      jTwiddle = 0;
      for(j=start;j<PARAM_N-1;j+=2*distance)
      {
        W = omega[jTwiddle++];
        temp = a[j];
        a[j] = barrett_reduce((temp + a[j + distance]));
        a[j + distance] = montgomery_reduce((W * ((uint32_t)temp
+ 3*PARAM_Q - a[j + distance])));
      }
    }
  }
}
```

This function makes use of the Montgomery reduction and the Barrett reduction function, which are defined in `reduce.c`.

```
#include "reduce.h"
#include "params.h"

/* Incomplete-reduction routines; for details on allowed input ranges
 * and produced output ranges, see the description in the paper:
 * https://cryptojedi.org/papers/#newhope */


static const uint32_t qinv = 12287; // -inverse_mod(p,2^18)
static const uint32_t rlog = 18;

uint16_t montgomery_reduce(uint32_t a)
{
  uint32_t u;

  u = (a * qinv);
  u &= ((1<<rlog)-1);
  u *= PARAM_Q;
  a = a + u;
  return a >> 18;
}


uint16_t barrett_reduce(uint16_t a)
{
  uint32_t u;

  u = ((uint32_t) a * 5) >> 16;
  u *= PARAM_Q;
  a -= u;
  return a;
}
```

Our own optimisation, `ntt_parallel.c` already includes the parellised Montgomery reduction and Barrett reduction. Also, the loop is unrolled in order to simplify the debugging.

```
#include <stdio.h>
#include "inttypes.h"

#include "cvmx.h"
#include "cvmx-spinlock.h"
#include "cvmx-fpa.h"
#include "cvmx-ilk.h"
#include "cvmx-pip.h"
#include "cvmx-ipd.h"
#include "cvmx-pko.h"
#include "cvmx-dfa.h"
#include "cvmx-pow.h"
#include "cvmx-gmx.h"
#include "cvmx-sysinfo.h"
#include "cvmx-coremask.h"
#include "cvmx-bootmem.h"
#include "cvmx-helper.h"
#include "cvmx-app-hotplug.h"
#include "cvmx-helper-cfg.h"
#include "cvmx-srio.h"
#include "cvmx-rng.h"

#define PARAM_Q 12289
#define PARAM_N 1024

#define RLOG 18
#define R_LOG_1 262143
#define QINV 12287

static const uint32_t qinv = 12287; // -inverse_mod(p,2^18)
static const uint32_t rlog = 18;

uint16_t montgomery_reduce(uint32_t a)
{
  uint32_t u;
  u = (a * qinv);
  u &= ((1<<rlog)-1);
  u *= PARAM_Q;
  a = a + u;
  return a >> 18;
}
uint64_t barrett_reduce64(uint64_t a)
{
  uint64_t u;

  u = a;
  u = u * 5;
  u = u & 0xFFFF0000FFFF0000;
  u = u >> 16;
  u = u * PARAM_Q;
  a = a - u;
  return a;
}

uint64_t montgomery_reduce64(uint64_t a)
{
  uint64_t u,u1,u2;
  u1 = a & 0xFFFFFFFF00000000;
  u1 *= qinv;
  u2 = a & 0x00000000FFFFFFFF;
  u2 *= qinv;
  u1 = u1 & 0x0003FFFF00000000;
  u2 = u2 & 0x000000000003FFFF;
  u = u1 + u2;
  u = u * PARAM_Q;
  a = a + u;
  //  a = a & 0xFFFC0000FFFC0000;
  return a >> 18;
}


void ntt_parallel(uint16_t * a, uint16_t* omega)
{
  int i, start, j, jTwiddle, distance;
  uint16_t W;
  uint64_t temp;
  uint64_t aa[512];
  //Even level 0 (not parallel)
  distance = (1<<0);
  for(start = 0; start < distance;start++)
  {
    jTwiddle = 0;
    for(j=start;j<PARAM_N-1;j+=2*distance)
    {
      W = omega[jTwiddle++];
      temp = a[j];
      a[j] = (temp + a[j + distance]); // Omit reduction (be lazy)
      a[j + distance] = montgomery_reduce((W * ((uint32_t)temp +
3*PARAM_Q - a[j + distance])));
    }
  }

  //Translate from 16-bit integers to 64-bit integers.
  for(i=0;i<PARAM_N/2;i++)
    aa[i] = a[2*i+1] | (uint64_t)a[2*i]<<32;

  //Odd level 1
  distance = (1 << (1-1)); //Only half (2 coeff. in 1 register ==>
distance in table gets twice as small)
  for(start = 0; start < distance;start++)
  {
    jTwiddle = 0;
    for(j=start;j<PARAM_N/2-distance;j+=2*distance)
    {
      W = omega[jTwiddle++];
      temp = aa[j];
      aa[j] = barrett_reduce64((temp + aa[j + distance]));
      aa[j + distance] = montgomery_reduce64((W * (temp + 0x0000900300009003
- aa[j + distance])));
    }
  }

  //Even level 2
  distance = (1 << (2-1));
  for(start = 0; start < distance;start++)
  {
    jTwiddle = 0;
    for(j=start;j<PARAM_N/2-distance;j+=2*distance)
    {
      W = omega[jTwiddle++];
      temp = aa[j];
      aa[j] = (temp + aa[j + distance]);
      aa[j + distance] = montgomery_reduce64((W * (temp + 0x0000900300009003
- aa[j + distance])));
    }
  }

  //Odd level 3
  distance = (1 << (3-1));
  for(start = 0; start < distance;start++)
  {
    jTwiddle = 0;
    for(j=start;j<PARAM_N/2-distance;j+=2*distance)
    {
      W = omega[jTwiddle++];
      temp = aa[j];
      aa[j] = barrett_reduce64((temp + aa[j + distance]));
      aa[j + distance] = montgomery_reduce64((W * (temp + 0x0000900300009003
- aa[j + distance])));
    }
  }

  //Even level 4
  distance = (1 << (4-1));
  for(start = 0; start < distance;start++)
  {
    jTwiddle = 0;
    for(j=start;j<PARAM_N/2-distance;j+=2*distance)
    {
      W = omega[jTwiddle++];
      temp = aa[j];
      aa[j] = (temp + aa[j + distance]);
      aa[j + distance] = montgomery_reduce64((W * (temp + 0x0000900300009003
- aa[j + distance])));
    }
  }

  //Odd level 5
  distance = (1 << (5-1));
  for(start = 0; start < distance;start++)
  {
    jTwiddle = 0;
    for(j=start;j<PARAM_N/2-distance;j+=2*distance)
    {
      W = omega[jTwiddle++];
      temp = aa[j];
      aa[j] = barrett_reduce64((temp + aa[j + distance]));
      aa[j + distance] = montgomery_reduce64((W * (temp + 0x0000900300009003
- aa[j + distance])));
    }
  }

  //Even level 6
  distance = (1 << (6-1));
  for(start = 0; start < distance;start++)
  {
    jTwiddle = 0;
    for(j=start;j<PARAM_N/2-distance;j+=2*distance)
    {
      W = omega[jTwiddle++];
      temp = aa[j];
      aa[j] = (temp + aa[j + distance]);
      aa[j + distance] = montgomery_reduce64((W * (temp + 0x0000900300009003
- aa[j + distance])));
    }
  }


  //Odd level 7
  distance = (1 << (7-1));
  for(start = 0; start < distance;start++)
  {
    jTwiddle = 0;
    for(j=start;j<PARAM_N/2-distance;j+=2*distance)
    {
      W = omega[jTwiddle++];
      temp = aa[j];
      aa[j] = barrett_reduce64((temp + aa[j + distance]));
      aa[j + distance] = montgomery_reduce64((W * (temp + 0x0000900300009003
- aa[j + distance])));
    }
  }

  //Even level 8
  distance = (1 << (8-1));
  for(start = 0; start < distance;start++)
  {
    jTwiddle = 0;
    for(j=start;j<PARAM_N/2-distance;j+=2*distance)
    {
      W = omega[jTwiddle++];
      temp = aa[j];
      aa[j] = (temp + aa[j + distance]);
      aa[j + distance] = montgomery_reduce64((W * (temp + 0x0000900300009003
- aa[j + distance])));
    }
  }

  //Odd level 9
  distance = (1 << (9-1));
  for(start = 0; start < distance;start++)
  {
    jTwiddle = 0;
    for(j=start;j<PARAM_N/2-distance;j+=2*distance)
    {
      W = omega[jTwiddle++];
      temp = aa[j];
      aa[j] = barrett_reduce64((temp + aa[j + distance]));
      aa[j + distance] = montgomery_reduce64((W * (temp + 0x0000900300009003
- aa[j + distance])));
    }
  }

  //Translate back to 16-bit integers
  for(i=0;i<PARAM_N/2;i++)
  {
    a[2*i]   = aa[i] >> 32;
    a[2*i+1] = aa[i] & 0xffffffff;
  }
}
```

B NHS optimisation in (MIPS64 assembly)

The MIPS64 assembly code is divided into 4 files. In `pnttlevel0.s`, we deal with the first layer. In `pnttlevel1t4.s`, layer 1 up to and including 4 are implemented. `pnttlevel5t8.s` contains layer 5, 6, 7, and 8. The last layer is included in `pnttlevel9.s`.

We start with `pnttlevel0.s`.

```
.text
.globl pnttlevel0
.ent pnttlevel0

.macro MONTG_0 INPUT
  li  $v0, 12287 /* *QINV */
  mul  $gp, \INPUT, $v0
  ext $gp, $gp, 0, 18
  li  $v0, 12289 /* *PARAM_Q */
  mul  $gp, $gp, $v0
  addu $gp, $gp, \INPUT /* a + u */
  ext \INPUT, $gp, 18, 14
.endm

.macro SUBROUTINE_0 COEFF1 COEFF2 OFFSETOM OFFSETA
  lh  $t8, \OFFSETOM($a1) #$t8 bevat nu omega
  add $t9, \COEFF1, \COEFF2 #a[j] = a[j] + a[j + distance]
  li  $v0, 36867
  add \COEFF1, \COEFF1, $v0 #temp += 3q
  sub \COEFF1, \COEFF1, \COEFF2 #temp -= $t1
  mul \COEFF1, \COEFF1, $t8 #temp *= W
  MONTG_0 \COEFF1
  dsll $t9, 32
  dadd  $t9, $t9, \COEFF1
  sd    $t9, \OFFSETA($a0)
.endm

.macro ROUTINE_0
  SUBROUTINE_0 $t0, $t1,  0,  0
  SUBROUTINE_0 $t2, $t3,  2,  8
  SUBROUTINE_0 $t4, $t5,  4, 16
  SUBROUTINE_0 $t6, $t7,  6, 24
  SUBROUTINE_0 $s0, $s1,  8, 32
  SUBROUTINE_0 $s2, $s3, 10, 40
  SUBROUTINE_0 $s4, $s5, 12, 48
  SUBROUTINE_0 $s6, $s7, 14, 56
.endm

.macro ROUTINE_1
  SUBROUTINE_0 $t0, $t1, 16,  64
  SUBROUTINE_0 $t2, $t3, 18,  72
  SUBROUTINE_0 $t4, $t5, 20,  80
  SUBROUTINE_0 $t6, $t7, 22,  88
  SUBROUTINE_0 $s0, $s1, 24,  96
  SUBROUTINE_0 $s2, $s3, 26, 104
  SUBROUTINE_0 $s4, $s5, 28, 112
  SUBROUTINE_0 $s6, $s7, 30, 120
.endm

.macro LOAD_0
     lh $t0,   0($a0)
     lh $t1,   2($a0)
     lh $t2,   4($a0)
     lh $t3,   6($a0)
     lh $t4,   8($a0)
     lh $t5,  10($a0)
     lh $t6,  12($a0)
     lh $t7,  14($a0)
     lh $s0,  16($a0)
     lh $s1,  18($a0)
     lh $s2,  20($a0)
     lh $s3,  22($a0)
     lh $s4,  24($a0)
     lh $s5,  26($a0)
     lh $s6,  28($a0)
     lh $s7,  30($a0)
.endm

.macro LOADSTORE_NEW
  ld $t0,  2048($a0)
  sd    $t0,  0($a0)
  ld $t1,  2056($a0)
  sd    $t1,  8($a0)
  ld $t2,  2064($a0)
  sd    $t2,  16($a0)
  ld $t3,  2072($a0)
  sd    $t3,  24($a0)
  ld $t4,  2080($a0)
  sd    $t4,  32($a0)
  ld $t5,  2088($a0)
  sd    $t5,  40($a0)
  ld $t6,  2096($a0)
  sd    $t6,  48($a0)
  ld $t7,  2104($a0)
  sd    $t7,  56($a0)
.endm

.macro LOAD_SP
  addi $sp, $sp, -96
  sd $s0,  0($sp)
  sd $s1,  8($sp)
  sd $s2, 16($sp)
  sd $s3, 24($sp)
  sd $s4, 32($sp)
  sd $s5, 40($sp)
  sd $s6, 48($sp)
  sd $s7, 56($sp)
  sd $gp, 64($sp)
  sd $fp, 72($sp)
  sd $ra, 80($sp)
.endm

.macro STORE_SP
  ld  $s0,  0($sp)
  ld  $s1,  8($sp)
  ld  $s2, 16($sp)
  ld  $s3, 24($sp)
  ld  $s4, 32($sp)
  ld  $s5, 40($sp)
  ld  $s6, 48($sp)
  ld  $s7, 56($sp)
  ld  $gp, 64($sp)
  ld  $fp, 72($sp)
  ld  $ra, 80($sp)
  addi $sp, $sp, 88
```

```
.endm

.macro TEST_OM
  addi $a1, $a1, 16
  lh $t0, 0($a1)
  lh $t1, 2($a1)
  lh $t2, 4($a1)
  lh $t3, 6($a1)
  lh $t4, 16($a1)
  lh $t5, 18($a1)
  lh $t6, 20($a1)
  lh $t7, 22($a1)
  sh $t0, 0($a0)
  sh $t1, 2($a0)
  sh $t2, 4($a0)
  sh $t3, 6($a0)
  sh $t4, 8($a0)
  sh $t5, 10($a0)
  sh $t6, 12($a0)
  sh $t7, 14($a0)
.endm

.macro LOADSTORE_1
  LOADSTORE_NEW
  addi $a0, $a0, 64
  LOADSTORE_NEW
  addi $a0, $a0, 64
  LOADSTORE_NEW
  addi $a0, $a0, 64
  LOADSTORE_NEW
  addi $a0, $a0, 64
  LOADSTORE_NEW
  addi $a0, $a0, 64
  LOADSTORE_NEW
  addi $a0, $a0, 64
  LOADSTORE_NEW
  addi $a0, $a0, 64
  LOADSTORE_NEW
  addi $a0, $a0, 64
.endm
```

```
.macro LOADSTORE_2
  LOADSTORE_1
  LOADSTORE_1
  LOADSTORE_1
  LOADSTORE_1
  LOADSTORE_1
  LOADSTORE_1
  LOADSTORE_1
  LOADSTORE_NEW
  addi $a0, $a0, 64
  LOADSTORE_NEW
  addi $a0, $a0, 64
  LOADSTORE_NEW
  addi $a0, $a0, 64
  LOADSTORE_NEW
  addi $a0, $a0, 64
  LOADSTORE_NEW
  addi $a0, $a0, 64
  LOADSTORE_NEW
  addi $a0, $a0, 64
  LOADSTORE_NEW
  addi $a0, $a0, 64
  LOADSTORE_NEW
.endm

  pnttlevel0:
  /* Init */

  LOAD_SP

  #initialise counter
  li $ra, 0
  sd $ra, 88($sp)

  looptop:

  #Shift $a0 with 32*i
  ld    $ra, 88($sp)
  li $v1, 32
  mul $v1, $v1, $ra
  add  $a0, $a0, $v1
```

```
LOAD_0

#Shift, such that you don’t overwrite the previous results
addi $a0, $a0, 2048
add $a0, $a0, $v1

ROUTINE_0 #apply operations and store behind the 2048$(a0)
addi $a0, $a0, -2048
sub $a0, $a0, $v1
sub $a0, $a0, $v1

/* shift omegas */
addi $a1, $a1, 16 #shift omegas

#Increment loop
ld    $ra, 88($sp)
addi  $ra, 1 #loop counter += 1
sd    $ra, 88($sp)
li    $v1, 64

bnel $ra, $v1, looptop

#Afterwards: store values on right place
LOADSTORE_2

STORE_SP

jr $ra

.end pnttlevel0
```

Next, we see that `pnttlevel1t4.s` contains:

```
.text
.globl pnttlevel1t4
.ent pnttlevel1t4

/* Registers:
   $v0 : PARAM_Q|0000|QINV
   $v1 : temp for loop counter and computations on PARAM_Q and QINV
   $a0 : address(a)
   $a1 : address(omega)
   $a2 : 0x0000900300009003
   $a3 : 0x0000FFFF0000FFFF ==> OR temp montgomery ($t9)
   $t0 - $t7 : coefficients a (2 per register)
   $s0 - $s7 : coefficients a (2 per register)
   $t8 : omegas (4 per register)
   $t9 : temp
   $gp : temp (barrett and montgomery)
   $fp : 0x00000000ffffffff
   $ra : temp (loop counter pushed to stack)
 */

/* Uses $gp as temp space */
.macro BARRETT_REDUCE OUT IN
  dsll  $gp,\IN,2
  daddu $gp,$gp,\IN
  dsrl $gp, $gp, 16
  and $gp, $gp, $a3
  #dsrl $v1, $v0, 32
  li $v0, 12289
  dmul $gp, $gp, $v0  #put $v1 = 12289
  dsub \OUT, \IN, $gp
.endm

/* Uses $t9, $gp, and $ra as temp space */
.macro MONTGOMERY_REDUCE INOUT
  dsll    $gp, $fp, 32
  and     $t9, \INOUT, $fp
  and     $gp, \INOUT, $gp
  #andi $v1, $v0, 0xFFFF #put $v1 = 12287
  li $v0, 12287
  dmul    $t9, $t9, $v0
  dmul    $gp, $gp, $v0
  dsrl    $ra, $fp, 14
  and     $t9, $t9, $ra
  dsll    $ra, $ra, 32
  and     $gp, $gp, $ra
  daddu   $t9, $t9, $gp
  #dsrl $v1, $v0, 32 #put $v1 = 12289
  li $v0, 12289
  dmul $t9, $t9, $v0
  daddu \INOUT, \INOUT, $t9
  dsrl \INOUT, \INOUT, 18
.endm

#WITH Barrett_reduction
.macro SUBROUTINE_B COEF1 COEF2 SH1 SH2 SH3
  lh $t8, \SH1($a1) #load 1st omega

  dadd  $t9, \COEF1, \COEF2  #(temp + aa[j + distance])

  dadd \COEF1, \COEF1, $a2 #temp += 3*PARAM_Q
  dsub \COEF2, \COEF1, \COEF2 #-= aa[j + distance]
  dmul \COEF2, \COEF2, $t8 #*= W

  BARRETT_REDUCE \COEF1, $t9
  #sd \COEF1, \SH2($a0) #Going back from temp value t9 to t0

  MONTGOMERY_REDUCE \COEF2
  #sd \COEF2, \SH3($a0)
.endm

  #WITHOUT Barrett_reduction
.macro SUBROUTINE COEF1 COEF2 SH1 SH2 SH3
  lh $t8, \SH1($a1) #load omega

  dadd  $t9, \COEF1, \COEF2  #(temp + aa[j + distance])

  dadd \COEF1, \COEF1, $a2 #temp += 3*PARAM_Q
  dsub \COEF2, \COEF1, \COEF2 #-= aa[j + distance]
  dmul \COEF2, \COEF2, $t8 #*= W
  and \COEF1, $t9, $t9 #instead of using sd

  #sd $t9, \SH2($a0) #Going back from temp value t9 to t0

  MONTGOMERY_REDUCE \COEF2
  #sd \COEF2, \SH3($a0)
.endm

.macro SUBROUTINE_0 COEF1 COEF2 SH1 SH2 SH3
  lh $t8, \SH1($a1) #load omega

  dadd  $t9, \COEF1, \COEF2  #(temp + aa[j + distance])

  li $v0, 36867
  add \COEF1, \COEF1, $v0 #temp += 3*PARAM_Q
  sub \COEF2, \COEF1, \COEF2 #-= aa[j + distance]
  mul \COEF2, \COEF2, $t8 #*= W

  MONTGOMERY_REDUCE \COEF2 #aanpassen naar ander formaat
  dsll $t9, $t9, 32
  dadd $t9, $t9, \COEF2
  sd $t9, \SH2($a0) #Going back from temp value t9 to t0
.endm

.macro LAYER1
  SUBROUTINE_B $t0, $t1, 0, 0, 8
  SUBROUTINE_B $t2, $t3, 2, 16, 24
  SUBROUTINE_B $t4, $t5, 4, 32, 40
  SUBROUTINE_B $t6, $t7, 6, 48, 56
  SUBROUTINE_B $s0, $s1, 8, 64, 72
  SUBROUTINE_B $s2, $s3, 10, 80, 88
  SUBROUTINE_B $s4, $s5, 12, 96, 104
  SUBROUTINE_B $s6, $s7, 14, 112, 120
.endm

.macro LAYER2
  SUBROUTINE $t0, $t2, 0, 0, 16
  SUBROUTINE $t1, $t3, 0, 8, 24
  SUBROUTINE $t4, $t6, 2, 32, 48
  SUBROUTINE $t5, $t7, 2, 40, 56
  SUBROUTINE $s0, $s2, 4, 64, 80
  SUBROUTINE $s1, $s3, 4, 72, 88
  SUBROUTINE $s4, $s6, 6, 96, 112
  SUBROUTINE $s5, $s7, 6, 104, 120
.endm

.macro LAYER3
  SUBROUTINE_B $t0, $t4, 0, 0, 32
  SUBROUTINE_B $t1, $t5, 0, 8, 40
  SUBROUTINE_B $t2, $t6, 0, 16, 48
  SUBROUTINE_B $t3, $t7, 0, 24, 56
  SUBROUTINE_B $s0, $s4, 2, 64, 96
  SUBROUTINE_B $s1, $s5, 2, 72, 104
  SUBROUTINE_B $s2, $s6, 2, 80, 112
  SUBROUTINE_B $s3, $s7, 2, 88, 120
.endm

.macro LAYER4
  SUBROUTINE $t0, $s0, 0, 0, 64
  SUBROUTINE $t1, $s1, 0, 8, 72
  SUBROUTINE $t2, $s2, 0, 16, 80
  SUBROUTINE $t3, $s3, 0, 24, 88
  SUBROUTINE $t4, $s4, 0, 32, 96
  SUBROUTINE $t5, $s5, 0, 40, 104
  SUBROUTINE $t6, $s6, 0, 48, 112
  SUBROUTINE $t7, $s7, 0, 56, 120
.endm

.macro LOAD
  ld $t0,   0($a0)
  ld $t1,   8($a0)
  ld $t2,  16($a0)
  ld $t3,  24($a0)
  ld $t4,  32($a0)
  ld $t5,  40($a0)
  ld $t6,  48($a0)
  ld $t7,  56($a0)
  ld $s0,  64($a0)
  ld $s1,  72($a0)
  ld $s2,  80($a0)
  ld $s3,  88($a0)
  ld $s4,  96($a0)
  ld $s5, 104($a0)
  ld $s6, 112($a0)
  ld $s7, 120($a0)
.endm

  pnttlevel1t4:
  /* Init */

  addi $sp, $sp, -96
  sd $s0,  0($sp)
  sd $s1,  8($sp)
  sd $s2, 16($sp)
  sd $s3, 24($sp)
  sd $s4, 32($sp)
  sd $s5, 40($sp)
  sd $s6, 48($sp)
  sd $s7, 56($sp)
  sd $gp, 64($sp)
  sd $fp, 72($sp)
  sd $ra, 80($sp)

  li $v0, 12289
  dsll $v0, $v0, 32
  daddi $v0, $v0, 12287 #$v0 now holds 0x300100002fff (=12289|0000|12287)

  li $a2, 0x9003
  dsll $ra, $a2, 32
  dadd $a2, $a2, $ra # $a2 now holds 0x0000900300009003

  li $a3, 0xFFFF
  dsll $ra, $a3, 32
  dadd $a3, $a3, $ra # $a3 now holds 0x0000FFFF0000FFFF

  li $ra, 0  #initialise counter
  sd $ra, 88($sp)

  li $fp, -1
  dsrl $fp, $fp, 32

  looptop:
```

```
LOAD

ld    $ra, 88($sp)
li $v1, 16
mul $v1, $v1, $ra
add  $a1, $a1, $v1 #Move to next omega
LAYER1
sub $a1, $a1, $v1   #Move the address back

ld    $ra, 88($sp)
li $v1, 8
mul $v1, $v1, $ra
add  $a1, $a1, $v1 #Move to next omega
LAYER2
sub $a1, $a1, $v1   #Move the address back

ld    $ra, 88($sp)
li $v1, 4
mul $v1, $v1, $ra
add  $a1, $a1, $v1 #Move to next omega
LAYER3
sub $a1, $a1, $v1   #Move the address back

ld    $ra, 88($sp)
li $v1, 2
mul $v1, $v1, $ra
add  $a1, $a1, $v1 #Move to next omega
LAYER4
sub $a1, $a1, $v1   #Move the address back

#STORE VARIABLES
sd $t0,   0($a0)
sd $t1,   8($a0)
sd $t2,  16($a0)
sd $t3,  24($a0)
sd $t4,  32($a0)
sd $t5,  40($a0)
sd $t6,  48($a0)
sd $t7,  56($a0)
sd $s0,  64($a0)
sd $s1,  72($a0)
sd $s2,  80($a0)
sd $s3,  88($a0)
sd $s4,  96($a0)
sd $s5, 104($a0)
sd $s6, 112($a0)
sd $s7, 120($a0)

addi  $a0, $a0, 128 #Move to next block of a's
ld    $ra, 88($sp)
addi  $ra, 1 #loop counter += 1
sd    $ra, 88($sp)
li    $v1, 32

bnel $ra, $v1, looptop

ld $s0,  0($sp)
ld $s1,  8($sp)
ld $s2, 16($sp)
ld $s3, 24($sp)
ld $s4, 32($sp)
ld $s5, 40($sp)
ld $s6, 48($sp)
ld $s7, 56($sp)
ld $gp, 64($sp)
ld $fp, 72($sp)
ld $ra, 80($sp)
addi $sp, $sp, 88

jr $ra

.end pnttlevel1t4
```

Afterwards, we perform `pnttlevel5t8.s`, which is defined as:

```
.text
.globl pnttlevel5t8
.ent pnttlevel5t8

/* Registers:
   $v0 : PARAM_Q|0000|QINV
   $v1 : temp for loop counter and computations on PARAM_Q and QINV
   $a0 : address(a)
   $a1 : address(omega)
   $a2 : 0x0000900300009003
   $a3 : 0x0000FFFF0000FFFF ==> OR temp montgomery ($t9)
   $t0 - $t7 : coefficients a (2 per register)
   $s0 - $s7 : coefficients a (2 per register)
   $t8 : omegas (4 per register)
   $t9 : temp
   $gp : temp (barrett and montgomery)
   $fp : 0x00000000ffffffff
   $ra : temp (loop counter pushed to stack)
 */

/* Uses $gp as temp space */
.macro BARRETT_REDUCE OUT IN
  dsll  $gp,\IN,2
  daddu $gp,$gp,\IN
  dsrl $gp, $gp, 16
  and $gp, $gp, $a3
  #dsrl $v1, $v0, 32
  li $v0, 12289
  dmul $gp, $gp, $v0  #put $v1 = 12289
  dsub \OUT, \IN, $gp
.endm

/* Uses $t9, $gp, and $ra as temp space */
.macro MONTGOMERY_REDUCE INOUT
  dsll    $gp, $fp, 32
  and     $t9, \INOUT, $fp
  and     $gp, \INOUT, $gp
  #andi $v1, $v0, 0xFFFF #put $v1 = 12287
  li $v0, 12287
  dmul    $t9, $t9, $v0
  dmul    $gp, $gp, $v0
  dsrl    $ra, $fp, 14
  and     $t9, $t9, $ra
  dsll    $ra, $ra, 32
  and     $gp, $gp, $ra
  daddu   $t9, $t9, $gp
  #dsrl $v1, $v0, 32 #put $v1 = 12289
  li $v0, 12289
  dmul $t9, $t9, $v0
  daddu \INOUT, \INOUT, $t9
  dsrl \INOUT, \INOUT, 18
.endm

#WITH Barrett_reduction
.macro SUBROUTINE_B COEF1 COEF2 SH1
  lh $t8, \SH1($a1) #load 1st omega

  dadd  $t9, \COEF1, \COEF2  #(temp + aa[j + distance])

  dadd \COEF1, \COEF1, $a2 #temp += 3*PARAM_Q
  dsub \COEF2, \COEF1, \COEF2 #-= aa[j + distance]
  dmul \COEF2, \COEF2, $t8 #*= W

  BARRETT_REDUCE \COEF1, $t9

  MONTGOMERY_REDUCE \COEF2
.endm

  #WITHOUT Barrett_reduction
.macro SUBROUTINE COEF1 COEF2 SH1 SH2 SH3
  lh $t8, \SH1($a1) #load omega

  dadd  $t9, \COEF1, \COEF2  #(temp + aa[j + distance])

  dadd \COEF1, \COEF1, $a2 #temp += 3*PARAM_Q
  dsub \COEF2, \COEF1, \COEF2 #-= aa[j + distance]
  dmul \COEF2, \COEF2, $t8 #*= W

  and \COEF1, $t9, $t9 #instead of using sd
  #sd $t9, \SH2($a0) #Going back from temp value t9 to t0

  MONTGOMERY_REDUCE \COEF2
  #sd \COEF2, \SH3($a0)
.endm

.macro LAYER5
  SUBROUTINE_B $t0, $t1, 0
  SUBROUTINE_B $t2, $t3, 2
  SUBROUTINE_B $t4, $t5, 4
  SUBROUTINE_B $t6, $t7, 6
  SUBROUTINE_B $s0, $s1, 8
  SUBROUTINE_B $s2, $s3, 10
  SUBROUTINE_B $s4, $s5, 12
  SUBROUTINE_B $s6, $s7, 14
.endm

.macro LAYER6
  SUBROUTINE $t0, $t2, 0
  SUBROUTINE $t1, $t3, 0
  SUBROUTINE $t4, $t6, 2
  SUBROUTINE $t5, $t7, 2
  SUBROUTINE $s0, $s2, 4
  SUBROUTINE $s1, $s3, 4
  SUBROUTINE $s4, $s6, 6
  SUBROUTINE $s5, $s7, 6
.endm

.macro LAYER7
  SUBROUTINE_B $t0, $t4, 0
  SUBROUTINE_B $t1, $t5, 0
  SUBROUTINE_B $t2, $t6, 0
  SUBROUTINE_B $t3, $t7, 0
  SUBROUTINE_B $s0, $s4, 2
  SUBROUTINE_B $s1, $s5, 2
  SUBROUTINE_B $s2, $s6, 2
  SUBROUTINE_B $s3, $s7, 2
.endm

.macro LAYER8
  SUBROUTINE $t0, $s0, 0
  SUBROUTINE $t1, $s1, 0
  SUBROUTINE $t2, $s2, 0
  SUBROUTINE $t3, $s3, 0
  SUBROUTINE $t4, $s4, 0
  SUBROUTINE $t5, $s5, 0
  SUBROUTINE $t6, $s6, 0
  SUBROUTINE $t7, $s7, 0
.endm

.macro LOAD_SEC
  ld $t0,    0($a0)
  ld $t1,  128($a0)
  ld $t2,  256($a0)
  ld $t3,  384($a0)
  ld $t4,  512($a0)
  ld $t5,  640($a0)
  ld $t6,  768($a0)
  ld $t7,  896($a0)
  ld $s0, 1024($a0)
  ld $s1, 1152($a0)
  ld $s2, 1280($a0)
  ld $s3, 1408($a0)
  ld $s4, 1536($a0)
  ld $s5, 1664($a0)
  ld $s6, 1792($a0)
  ld $s7, 1920($a0)
.endm

.macro STORE_SEC
  sd $t0,    0($a0)
  sd $t1,  128($a0)
  sd $t2,  256($a0)
  sd $t3,  384($a0)
  sd $t4,  512($a0)
  sd $t5,  640($a0)
  sd $t6,  768($a0)
  sd $t7,  896($a0)
  sd $s0, 1024($a0)
  sd $s1, 1152($a0)
  sd $s2, 1280($a0)
  sd $s3, 1408($a0)
  sd $s4, 1536($a0)
  sd $s5, 1664($a0)
  sd $s6, 1792($a0)
  sd $s7, 1920($a0)
.endm

.macro LAYER5_LOOP_FIR
  LOAD_SEC
  LAYER5
  LAYER6
  LAYER7
  LAYER8
  STORE_SEC
  addi  $a0, $a0, 8 #Move to next block of a's
.endm

.macro LAYER5_LOOP_SEC
  LOAD_SEC
  LAYER5
  addi $a1, $a1, -8
  LAYER6
  addi $a1, $a1, -4
  LAYER7
  addi $a1, $a1, -2
  LAYER8
  addi $a1, $a1, 14
  STORE_SEC
  addi  $a0, $a0, 8 #Move to next block of a's
.endm

.macro LAYER5_LOOP_LOOP_FIR
  LAYER5_LOOP_FIR
  LAYER5_LOOP_FIR
  LAYER5_LOOP_FIR
  LAYER5_LOOP_FIR
  LAYER5_LOOP_FIR
  LAYER5_LOOP_FIR
  LAYER5_LOOP_FIR
  LAYER5_LOOP_FIR
  LAYER5_LOOP_FIR
  LAYER5_LOOP_FIR
  LAYER5_LOOP_FIR
  LAYER5_LOOP_FIR
  LAYER5_LOOP_FIR
  LAYER5_LOOP_FIR
  LAYER5_LOOP_FIR
  LAYER5_LOOP_FIR
.endm

.macro LAYER5_LOOP_LOOP_SEC
  LAYER5_LOOP_SEC
  LAYER5_LOOP_SEC
  LAYER5_LOOP_SEC
  LAYER5_LOOP_SEC
  LAYER5_LOOP_SEC
  LAYER5_LOOP_SEC
  LAYER5_LOOP_SEC
  LAYER5_LOOP_SEC
  LAYER5_LOOP_SEC
  LAYER5_LOOP_SEC
  LAYER5_LOOP_SEC
  LAYER5_LOOP_SEC
  LAYER5_LOOP_SEC
  LAYER5_LOOP_SEC
  LAYER5_LOOP_SEC
  LAYER5_LOOP_SEC
.endm

  pnttlevel5t8:
  /* Init */

  addi $sp, $sp, -96
  sd $s0,  0($sp)
  sd $s1,  8($sp)
  sd $s2, 16($sp)
  sd $s3, 24($sp)
  sd $s4, 32($sp)
  sd $s5, 40($sp)
  sd $s6, 48($sp)
  sd $s7, 56($sp)
  sd $gp, 64($sp)
  sd $fp, 72($sp)
  sd $ra, 80($sp)

  li $v0, 12289
  dsll $v0, $v0, 32
  daddi $v0, $v0, 12287 #$v0 now holds 0x300100002fff (=12289|0000|12287)

  li $a2, 0x9003
  dsll $ra, $a2, 32
  dadd $a2, $a2, $ra # $a2 now holds 0x0000900300009003

  li $a3, 0xFFFF
  dsll $ra, $a3, 32
  dadd $a3, $a3, $ra # $a3 now holds 0x0000FFFF0000FFFF

  li $ra, 0  #initialise counter
  sd $ra, 88($sp)

  li $fp, -1
  dsrl $fp, $fp, 32 #belangrijk: dit is NIET de counter

  #LOOP:
  LAYER5_LOOP_LOOP_FIR
  addi $a0, $a0, 1920 #shift to lower half
  addi $a1, $a1, 16 #shift omegas
  LAYER5_LOOP_LOOP_SEC

  ld $s0,  0($sp)
  ld $s1,  8($sp)
  ld $s2, 16($sp)
  ld $s3, 24($sp)
  ld $s4, 32($sp)
  ld $s5, 40($sp)
  ld $s6, 48($sp)
  ld $s7, 56($sp)
  ld $gp, 64($sp)
  ld $fp, 72($sp)
  ld $ra, 80($sp)
  addi $sp, $sp, 88

  jr $ra
```

```
.end pnttlevel5t8
```

The final layer, `pnttlevel9.s` contains:

```
.text
.globl pnttlevel9
.ent pnttlevel9

/* Registers:
   $v0 : PARAM_Q|0000|QINV
   $v1 : temp for loop counter and computations on PARAM_Q and QINV
   $a0 : address(a)
   $a1 : address(omega)
   $a2 : 0x0000900300009003
   $a3 : 0x0000FFFF0000FFFF ==> OF temp montgomery ($t9)
   $t0 - $t7 : coefficients a (2 per register)
   $s0 - $s7 : coefficients a (2 per register)
   $t8 : omegas (4 per register)
   $t9 : temp
   $gp : temp (barrett and montgomery)
   $fp : 0x00000000ffffffff
   $ra : temp (loop counter pushed to stack)
 */

/* Uses $gp as temp space */
.macro BARRETT_REDUCE OUT IN
  dsll  $gp,\IN,2
  daddu $gp,$gp,\IN
  dsrl $gp, $gp, 16
  and $gp, $gp, $a3
  #dsrl $v1, $v0, 32
  li $v0, 12289
  dmul $gp, $gp, $v0  #put $v1 = 12289
  dsub \OUT, \IN, $gp
.endm

/* Uses $t9, $gp, and $ra as temp space */
.macro MONTGOMERY_REDUCE INOUT
  dsll    $gp, $fp, 32
  and     $t9, \INOUT, $fp
  and     $gp, \INOUT, $gp
  #andi $v1, $v0, 0xFFFF #put $v1 = 12287
  li $v0, 12287
  dmul    $t9, $t9, $v0
  dmul    $gp, $gp, $v0
  dsrl    $ra, $fp, 14
  and     $t9, $t9, $ra
  dsll    $ra, $ra, 32
  and     $gp, $gp, $ra
  daddu   $t9, $t9, $gp
  #dsrl $v1, $v0, 32 #put $v1 = 12289
  li $v0, 12289
  dmul $t9, $t9, $v0
  daddu \INOUT, \INOUT, $t9
  dsrl \INOUT, \INOUT, 18
.endm

#WITH Barrett_reduction
.macro SUBROUTINE_B COEF1 COEF2 SH1
  lh $t8, \SH1($a1) #load 1st omega

  dadd  $t9, \COEF1, \COEF2  #(temp + aa[j + distance])

  dadd \COEF1, \COEF1, $a2 #temp += 3*PARAM_Q
  dsub \COEF2, \COEF1, \COEF2 #-= aa[j + distance]
  dmul \COEF2, \COEF2, $t8 #*= W

  BARRETT_REDUCE \COEF1, $t9

  MONTGOMERY_REDUCE \COEF2
.endm

  #WITHOUT Barrett_reduction
.macro SUBROUTINE COEF1 COEF2 SH1 SH2 SH3
  lh $t8, \SH1($a1) #load omega

  dadd  $t9, \COEF1, \COEF2  #(temp + aa[j + distance])

  dadd \COEF1, \COEF1, $a2 #temp += 3*PARAM_Q
  dsub \COEF2, \COEF1, \COEF2 #-= aa[j + distance]
  dmul \COEF2, \COEF2, $t8 #*= W

  and \COEF1, $t9, $t9 #instead of using sd
  #sd $t9, \SH2($a0) #Going back from temp value t9 to t0

  MONTGOMERY_REDUCE \COEF2
  #sd \COEF2, \SH3($a0)
.endm

.macro LAYER9
  SUBROUTINE_B $t0, $t1, 0
  SUBROUTINE_B $t2, $t3, 0
  SUBROUTINE_B $t4, $t5, 0
  SUBROUTINE_B $t6, $t7, 0
  SUBROUTINE_B $s0, $s1, 0
  SUBROUTINE_B $s2, $s3, 0
  SUBROUTINE_B $s4, $s5, 0
  SUBROUTINE_B $s6, $s7, 0
.endm

.macro LOAD_SEC
  ld $t0,    0($a0)
  ld $t1,  2048($a0)
  ld $t2,  8($a0)
  ld $t3,  2056($a0)
  ld $t4,  16($a0)
  ld $t5,  2064($a0)
  ld $t6,  24($a0)
  ld $t7,  2072($a0)
  ld $s0, 32($a0)
  ld $s1, 2080($a0)
  ld $s2, 40($a0)
  ld $s3, 2088($a0)
  ld $s4, 48($a0)
  ld $s5, 2096($a0)
  ld $s6, 56($a0)
  ld $s7, 2104($a0)
.endm

.macro STORE_SEC
  sd $t0,    0($a0)
  sd $t1,  2048($a0)
  sd $t2,  8($a0)
  sd $t3,  2056($a0)
  sd $t4,  16($a0)
  sd $t5,  2064($a0)
  sd $t6,  24($a0)
  sd $t7,  2072($a0)
  sd $s0, 32($a0)
  sd $s1, 2080($a0)
  sd $s2, 40($a0)
  sd $s3, 2088($a0)
  sd $s4, 48($a0)
  sd $s5, 2096($a0)
  sd $s6, 56($a0)
  sd $s7, 2104($a0)
.endm


  pnttlevel9:
  /* Init */

  addi $sp, $sp, -96
  sd $s0,  0($sp)
  sd $s1,  8($sp)
  sd $s2, 16($sp)
  sd $s3, 24($sp)
  sd $s4, 32($sp)
  sd $s5, 40($sp)
  sd $s6, 48($sp)
  sd $s7, 56($sp)
  sd $gp, 64($sp)
  sd $fp, 72($sp)
  sd $ra, 80($sp)

  li $v0, 12289
  dsll $v0, $v0, 32
  daddi $v0, $v0, 12287 #$v0 now holds 0x300100002fff (=12289|0000|12287)

  li $a2, 0x9003
  dsll $ra, $a2, 32
  dadd $a2, $a2, $ra # $a2 now holds 0x0000900300009003

  li $a3, 0xFFFF
  dsll $ra, $a3, 32
```

```
dadd $a3, $a3, $ra # $a3 now holds 0x0000FFFF0000FFFF

li $ra, 0  #initialise counter
sd $ra, 88($sp)

li $fp, -1
dsrl $fp, $fp, 32

/* Loop */

LOAD_SEC
LAYER9
STORE_SEC
addi  $a0, $a0, 64 #Move to next block of a’s
LOAD_SEC
LAYER9
STORE_SEC
addi  $a0, $a0, 64 #Move to next block of a’s
LOAD_SEC
LAYER9
STORE_SEC
addi  $a0, $a0, 64 #Move to next block of a’s
LOAD_SEC
LAYER9
STORE_SEC
addi  $a0, $a0, 64 #Move to next block of a’s
LOAD_SEC
LAYER9
STORE_SEC
addi  $a0, $a0, 64 #Move to next block of a’s
LOAD_SEC
LAYER9
STORE_SEC
addi  $a0, $a0, 64 #Move to next block of a’s
LOAD_SEC
LAYER9
STORE_SEC
addi  $a0, $a0, 64 #Move to next block of a’s
LOAD_SEC
LAYER9
STORE_SEC
```

```
addi  $a0, $a0, 64 #Move to next block of a’s
LOAD_SEC
LAYER9
STORE_SEC
addi  $a0, $a0, 64 #Move to next block of a’s
LOAD_SEC
LAYER9
STORE_SEC
addi  $a0, $a0, 64 #Move to next block of a’s
LOAD_SEC
LAYER9
STORE_SEC
addi  $a0, $a0, 64 #Move to next block of a’s
LOAD_SEC
LAYER9
STORE_SEC
addi  $a0, $a0, 64 #Move to next block of a’s
LOAD_SEC
LAYER9
STORE_SEC
addi  $a0, $a0, 64 #Move to next block of a’s
LOAD_SEC
LAYER9
STORE_SEC
addi  $a0, $a0, 64 #Move to next block of a’s
LOAD_SEC
LAYER9
STORE_SEC
addi  $a0, $a0, 64 #Move to next block of a’s
LOAD_SEC
LAYER9
STORE_SEC
addi  $a0, $a0, 64 #Move to next block of a’s
LOAD_SEC
LAYER9
STORE_SEC
addi  $a0, $a0, 64 #Move to next block of a’s
LOAD_SEC
LAYER9
STORE_SEC
addi  $a0, $a0, 64 #Move to next block of a’s
```

```
LOAD_SEC
LAYER9
STORE_SEC
addi  $a0, $a0, 64 #Move to next block of a’s
LOAD_SEC
LAYER9
STORE_SEC
addi  $a0, $a0, 64 #Move to next block of a’s
LOAD_SEC
LAYER9
STORE_SEC
addi  $a0, $a0, 64 #Move to next block of a’s
LOAD_SEC
LAYER9
STORE_SEC
addi  $a0, $a0, 64 #Move to next block of a’s
LOAD_SEC
LAYER9
STORE_SEC
addi  $a0, $a0, 64 #Move to next block of a’s
LOAD_SEC
LAYER9
STORE_SEC
addi  $a0, $a0, 64 #Move to next block of a’s
LOAD_SEC
LAYER9
STORE_SEC
addi  $a0, $a0, 64 #Move to next block of a’s
LOAD_SEC
LAYER9
STORE_SEC
addi  $a0, $a0, 64 #Move to next block of a’s
LOAD_SEC
LAYER9
STORE_SEC
addi  $a0, $a0, 64 #Move to next block of a’s
LOAD_SEC
LAYER9
STORE_SEC
addi  $a0, $a0, 64 #Move to next block of a’s
LOAD_SEC
```

```
LAYER9
STORE_SEC
addi  $a0, $a0, 64 #Move to next block of a's
LOAD_SEC
LAYER9
STORE_SEC
addi  $a0, $a0, 64 #Move to next block of a's
LOAD_SEC
LAYER9
STORE_SEC
addi  $a0, $a0, 64 #Move to next block of a's
LOAD_SEC
LAYER9
STORE_SEC
addi  $a0, $a0, 64 #Move to next block of a's

ld $s0,  0($sp)
ld $s1,  8($sp)
ld $s2, 16($sp)
ld $s3, 24($sp)
ld $s4, 32($sp)
ld $s5, 40($sp)
ld $s6, 48($sp)
ld $s7, 56($sp)
ld $gp, 64($sp)
ld $fp, 72($sp)
ld $ra, 80($sp)
addi $sp, $sp, 88

jr $ra

.end pnttlevel9
```

C NHS simulation based on RLWR (in Mathematica)

The simulation is denoted in `LWRsimple.nb`.

```
(*100 runs: \[PlusMinus]35 sec.*)
ClearAll["Global‘*"];
q = 12289;
p = 64;
n = 1024;
res = {};
ntests = 100;
For[j = 0, j < ntests, j++,
 (*Alice*)
 a = RandomInteger[q - 1, n];(*Generate n random samples*)
 s = RandomVariate[BinomialDistribution[128, 0.5], n] -
   64;(*Generate n binomial samples; was 32, 16*)
 ax = x^Range[0, n - 1].a;(*Represent as polynomial*)
 sx = x^Range[0, n - 1].s;(*Represent as polynomial*)
 asx = Expand[ax*sx];(*Multiply polynomials*)
 asxMod = PolynomialMod[asx, {x^n + 1, q}]; (*Negacyclic convolution*)


 asCoef =
  CoefficientList[asxMod,
   x];(*List of coefficients instead of polynomial*)
 b1 = asCoef*(p/q);
 b = Mod[Round[b1], p];
 diffB = Abs[b1 - b];
 (*Bob*)
 t = RandomVariate[BinomialDistribution[128, 0.5], n] - 64;
 tx = x^Range[0, n - 1].t;(*Represent as polynomial*)
 atx = Expand[ax*tx];(*Multiply polynomials*)
 atxMod = PolynomialMod[atx, {x^n + 1, q}]; (*Negacyclic convolution*)


 atCoef = CoefficientList[atxMod, x];
 u1 = atCoef*(p/q);
 u = Mod[Round[u1], p];
 diffU = Abs[u1 - u];
 k = RandomChoice[{0, Floor[p/2]}, n/4];(*Already encoded*)
 k4 = Join[k, k, k, k];(*Repeat on 4 places*)
 bx = x^Range[0, n - 1].b;(*Represent as polynomial*)
 btx = Expand[bx*tx];(*Multiply polynomials*)
 btxMod = PolynomialMod[btx, {x^n + 1, q}]; (*Negacyclic convolution*)


 btCoef =
  CoefficientList[btxMod,
   x];(*List of coefficients instead of polynomial*)
 c1 = btCoef*(p/q);
 (*c2=Mod[Round[c1],p];*)
 c2 = Round[c1];
 diffC = Abs[c1 - c2];
 c = Mod[c2 + k4, p];
 (*Alice*)
 ux = x^Range[0, n - 1].u;(*Represent as polynomial*)
 usx = Expand[ux*sx];(*Multiply polynomials*)
 usxMod = PolynomialMod[usx, {x^n + 1, q}]; (*Negacyclic convolution*)


 usCoef =
  CoefficientList[usxMod,
   x];(*List of coefficients instead of polynomial*)
 v1 = usCoef*(p/q);
 v = Mod[Round[v1], p];
 (*v=Round[v1];*)
 diffV = Abs[v1 - v];
 c - v;
 (*kA =Mod[c-v+(p/4),p]-(p/4);*)
 kA = Mod[c - v, p];
 \[Nu] = {};
 For[i = 1, i <= (n/4), i++,
   If[Abs[kA[[i]] - Floor[(p/2)]] +
      Abs[kA[[(n/4) + i]] - Floor[(p/2)]] +
      Abs[kA[[2*(n/4) + i]] - Floor[(p/2)]] +
      Abs[kA[[3*(n/4) + i]] - Floor[(p/2)]] < p, AppendTo[\[Nu],
1],
    AppendTo[\[Nu], 0]]](*Retrieve \[Nu]'*)
  AppendTo[res, Total[Abs[\[Nu] - (k/(p/2))]]];
 ]
If[Total[res] > 0, Histogram[res, p/2]
 , Print["No errors."]]
```

The debug.nb file looks as follows.

```
(*ClearAll["Global`*"];
q=12289;
p=16;
n=1024;
res={};
ntests=0;
While[Total[res]==0,
(*Alice*)
a = RandomInteger[q-1,n];(*Generate n random samples*)
s = RandomVariate[BinomialDistribution[32,0.5],n]-16;(*Generate n
\
binomial samples; was 32, 16*)
ax=x^Range[0,n-1].a;(*Represent as polynomial*)
sx=x^Range[0,n-1].s;(*Represent as polynomial*)
asx=Expand[ax*sx];(*Multiply polynomials*)
asxMod=PolynomialMod[asx,{x^n+1,q}]; (*Negacyclic convolution*)
asCoef = CoefficientList[asxMod,x];(*List of coefficients instead
of \
polynomial*)
b1=asCoef*(p/q);
b=Mod[Round[b1],p];
diffB=Abs[b1-b];
(*Bob*)
t= RandomVariate[BinomialDistribution[32,0.5],n]-16;
tx=x^Range[0,n-1].t;(*Represent as polynomial*)
atx=Expand[ax*tx];(*Multiply polynomials*)
atxMod=PolynomialMod[atx,{x^n+1,q}]; (*Negacyclic convolution*)
atCoef = CoefficientList[atxMod,x];
u1=atCoef*(p/q);
u=Mod[Round[u1],p];
diffU=Abs[u1-u];
k= RandomChoice[{0,Floor[p/2]},n/4];(*Already encoded*)
k4=Join[k,k,k,k];(*Repeat on 4 places*)
bx=x^Range[0,n-1].b;(*Represent as polynomial*)
btx=Expand[bx*tx];(*Multiply polynomials*)
btxMod=PolynomialMod[btx,{x^n+1,q}]; (*Negacyclic convolution*)
btCoef = CoefficientList[btxMod,x];(*List of coefficients instead
of \
polynomial*)
c1=btCoef*(p/q);
(*c2=Mod[Round[c1],p];*)
c2=Round[c1];
diffC=Abs[c1-c2];
c=Mod[c2+k4,p];
(*Alice*)
ux=x^Range[0,n-1].u;(*Represent as polynomial*)
usx=Expand[ux*sx];(*Multiply polynomials*)
usxMod=PolynomialMod[usx,{x^n+1,q}]; (*Negacyclic convolution*)
usCoef = CoefficientList[usxMod,x];(*List of coefficients instead
of \
polynomial*)
v1=usCoef*(p/q);
v=Mod[Round[v1],p];
(*v=Round[v1];*)
diffV=Abs[v1-v];
c-v;
(*kA =Mod[c-v+(p/4),p]-(p/4);*)
kA =Mod[c-v,p];
\[Nu]={};
For[i=1,i<=(n/4),i++,If[Abs[kA[[i]]-Floor[(p/2)]]+Abs[kA[[(n/4)+i]]-\
Floor[(p/2)]]+Abs[kA[[2*(n/4)+i]]-Floor[(p/2)]]+Abs[kA[[3*(n/4)+i]]-\
Floor[(p/2)]]<p,AppendTo[\[Nu],1],AppendTo[\[Nu],0]]](*Retrieve \
\[Nu]'*)
AppendTo[res,Total[Abs[\[Nu]-(k/(p/2))]]];
ntests++;
]
Print["Number of runs before first error: ",ntests,"; Absolute \
difference k and k':",Abs[\[Nu]-(k/(p/2))],"; Position(s) of non-zero
\
element(s): ",SparseArray[Abs[\[Nu]-(k/(p/2))]]["NonzeroPositions"]]*)

(*wr=286;
Print[(*"ax=",ax[[wr]],"; sx=",sx[[wr]],*)";asCoef=",asCoef[[wr]],";
\
b=",b [[wr]],"; diffB=",diffB[[wr]],(*"; tx=",tx[[wr]],*)"; \
atCoef=",atCoef[[wr]],"; u=",u[[wr]],"; diffU=",diffU[[wr]],"; \
k4=",k4[[wr]],(*"; bx=",bx[[wr]],*)"; btCoef=",btCoef[[wr]],"; \
c2=",c2[[wr]],"; diffC=",diffC[[wr]],"; c=",c[[wr]],(*"; \
ux=",ux[[wr]],*)"; usCoef=",usCoef[[wr]],"; v=",v[[wr]],"; \
diffV=",diffV[[wr]],"; c-v=",c[[wr]]-v[[wr]]]
```

```
(*Print["k'[i]=",kA[[wr]],"; k'[i+n/4]=",kA[[(n/4)+wr]],"; \
k'[i+2n/4]=",kA[[2*(n/4)+wr]],"; k'[i+3n/4]=",kA[[3*(n/4)+wr]]]*)*)

ClearAll["Global`*"];
q = 12289;
p = 4;
n = 1024;
hist = {};
histMod = {};
For[i = 0, i < 100, i++,
 a = RandomInteger[q - 1, n];
 s = RandomVariate[BinomialDistribution[32, 0.5], n] - 16;
 t = RandomVariate[BinomialDistribution[32, 0.5], n] - 16;
 (*s = RandomInteger[32,n]-16;
 t = RandomInteger[32,n]-16;*)(*Will give uniformly random output*)
 ax = x^Range[0, n - 1].a;
 sx = x^Range[0, n - 1].s;
 tx = x^Range[0, n - 1].t;
 asx = Expand[ax*sx];
 asxMod = PolynomialMod[asx, {x^n + 1, q}];
 asCoef = CoefficientList[asxMod, x];
 b1 = asCoef*(p/q);
 b = Mod[Round[b1], p];(*b=as*)
 bx = x^Range[0, n - 1].b;
 btx = Expand[bx*tx];
 btxMod = PolynomialMod[btx, {x^n + 1, q}];
 btCoef = CoefficientList[btxMod, x];
 c1 = btCoef*(p/q);
 c = Mod[Round[c1], p];(*c=ast*)
 atx = Expand[ax*tx];
 atxMod = PolynomialMod[atx, {x^n + 1, q}];
 atCoef = CoefficientList[atxMod, x];
 u1 = atCoef*(p/q);
 u = Mod[Round[u1], p];(*u=at*)
 ux = x^Range[0, n - 1].u;
 usx = Expand[ux*sx];
 usxMod = PolynomialMod[usx, {x^n + 1, q}];
 usCoef = CoefficientList[usxMod, x];
 v1 = usCoef*(p/q);
 v = Mod[Round[v1], p];(*v=ats*)
 AppendTo[hist, c - v];
 AppendTo[histMod, Abs[c - v]]]
(*Join[hist];*)
Histogram[hist]
Histogram[histMod]
```

The `debugUpdate.nb` file looks as follows.

```
(*Check the distributions of \[Epsilon] and s*)
ClearAll["Global‘*"];
nCoefs = 100000;(*Number of coefficients*)
q = 12289;
p = 12289/4;(*Divide by interval length*)
intLength = Floor[q/p];
\[Epsilon] = RandomInteger[intLength, nCoefs] - Floor[(intLength/2)];
s = RandomVariate[BinomialDistribution[32, 0.5], nCoefs] - 16;
Histogram[\[Epsilon]];
Histogram[s];
Histogram[\[Epsilon]*s];
\[Epsilon]x = x^Range[0, nCoefs - 1].\[Epsilon];
sx = x^Range[0, nCoefs - 1].s;
\[Epsilon]sx = Expand[\[Epsilon]x*sx];
\[Epsilon]sxMod = PolynomialMod[\[Epsilon]sx, {x^nCoefs + 1}];
\[Epsilon]sCoef = CoefficientList[\[Epsilon]sxMod, x];
min = Min[\[Epsilon]sCoef];
max = Max[\[Epsilon]sCoef];
\[Mu] = N[Mean[\[Epsilon]sCoef]];
\[Sigma] = N[StandardDeviation[\[Epsilon]sCoef]];
Histogram[\[Epsilon]sCoef]
Plot[PDF[NormalDistribution[\[Mu], \[Sigma]], x], {x, min, max}]
\[Mu]
\[Sigma]
min
max
```