Bachelor thesis
Computer Science & Mathematics

Radboud University

A New Hope for Nussbaumer

Author:
Gerben van der Lubbe
s4389026

First supervisor/assessor:
dr. P. Schwabe
peter@cryptojedi.org

Second assessor:
dr. L. Batina
lejla@cs.ru.nl

July 29, 2016

Abstract

In this thesis we focus on implementing an optimized version of Nussbaumer's negacyclic convolution algorithm for the New Hope key exchange protocol. To do so, we start by discussing the algorithm itself as specified by Knuth and the optimizations that can be applied to lower the amount of ring operations as specified by Nussbaumer. We shall see that the number-theoretic transform, the alternative algorithm New Hope uses to calculate the same, requires significantly fewer ring operations. Afterwards we build an AVX2 implementation of Nussbaumer's algorithm for New Hope, which allows us to tweak a constant chosen in the key exchange increasing security and furthermore simplifying reduction steps. We will then show that this turns out to not just increase the security of New Hope, but it also gives a more efficient implementation.

Contents

Chapter 1

Introduction

With the emerging of some basic quantum computers [1], even though they seem to provide a speedup of no more than a constant factor [2], the cryptographic scenery is changing. Many public-key ciphers that are commonly used, such as RSA and (elliptic curve) Diffie-Hellman, would efficiently be broken by more advanced quantum computers [3] through Shor's algorithm [4]; as such, new standards are actively considered for providing security in a post-quantum world [5].

One mathematical foundation for an algorithm that is efficient and furthermore thought to be secure even post-quantum is the Ring Learning with Errors (RLWE) problem [6]. Bos et al. proposed and implemented an instantiation of such in the form of a key exchange protocol using this problem [7]. Alkim et al. built on this paper, further improving and optimizing it [8].

Henceforth we will refer to the paper by Bos et al. as 'BCNS', after its authors, and the paper by Alkim et al. as 'New Hope', after its title. The latter shall be used interchangeably to mean either the paper, its described key exchange protocol or its accompanied implementation: its meaning should be clear from the context.

Due to the choice of parameters by BCNS, a common operation in the protocol is the multiplication of two polynomials in variable u with coefficients in $\mathbb{Z}/q\mathbb{Z}$, reduced modulo $u^n + 1$, where n and q are picked to be 1024 and $2^{32} - 1$, respectively; these multiplications turn out to be equivalent to the calculation of a negacyclic convolution. New Hope inherits these calculations, with the exception that a different q is adopted [8].

BCNS opted to use Nussbaumer's algorithm [9] in order to find the negacyclic convolution. They base their implementation on that covered in Knuth's 'the Art of Computer Programming' [10], exercise 4.6.4.59, even though "the algorithms are presented in unoptimized form" [10]. In fact, the few implementations of Nussbaumer's algorithm available are all based on the version by Knuth, and lack the implementation of further optimizations

Nussbaumer proposes in his paper.

New Hope introduces an improved analysis of the failure rate of a key exchange, which allows the lower value of $q = 12289$ to be chosen, "which improves both efficiency and security" [8]. The fact that q can not be decreased further is due to limitations of the number-theoretic transform (NTT) algorithm used instead of Nussbaumer's to calculate the polynomial product, which requires that $q \equiv 1 \pmod{2n}$. For this reason New Hope chose the value q to be minimal under these constraints.

1.1 Contributions

In this work we focus on Nussbaumer's algorithm, in particular in the context of New Hope. We provide the following:

- We give an in-depth overview of Nussbaumer's algorithm as described by Knuth [10], including its theoretic background and corresponding pseudo-code. We furthermore provide this version in the form of an unoptimized C++ implementation.
- We discuss and implement in C++ the optimizations presented by Nussbaumer in his paper [9], and consider their efficiency in terms of number of ring operations required for the algorithm. We compare the final implementation with that of the NTT implementation in New Hope using the same measure.
- We implement an optimized hybrid version (using Karatsuba's method for the recursion step) of Nussbaumer's algorithm using the AVX2 instruction set, in the context of New Hope. In particular, a different q is selected (no longer being subject to NTT's restrictions) in order to make the algorithm more secure and the reduction steps more efficient. We show that the implemented version actually beats the original NTT implementation not only in terms of security, but also in terms of performance.

The full source code is available at `https://github.com/spoofedex/nussbaumer`[1] and released under the public domain 'unlicense' [11].

We will not focus on perfecting our optimized version: we simply provide a version that is vectorized with thoroughly considered choices, even though further optimizations may and will still exist. Also note that we will

[1]The C++ implementation of Nussbaumer's algorithm is copied and slightly modified for each optimization, even though this seems to contradict the design principle to minimize duplicate code. We opted for this approach in order to prevent clutter of code for different versions, which would have a negative impact on readability, and in order to focus on the progressive nature of the distinct versions without making version control history an integral part of the project.

not go into depth on the cryptographic schemes associated to New Hope, as we merely focus on the aspect of negacyclic convolutions and are not immediately concerned with RLWE algorithms.

1.2 Related Work

Many algorithms exist to calculate a negacyclic convolution. First, it is possible to calculate a full polynomial multiplication, for example with Karatsuba's algorithm or the schoolbook algorithm, followed by a trivial reduction. However, there also exist specialized algorithms for calculating the negacyclic convolution, such as the number-theoretic transform methods, the Schönhage method and the Nussbaumer method [12].

We focus specifically on Nussbaumer's algorithm. Besides his own paper, the algorithm is explained both in exercise 4.6.4.59 of Knuth's 'the Art of Computer Programming' [10] and in the book 'Prime Numbers: A Computational Perspective' [12]. Neither of these produce the further optimizations presented in Nussbaumer's original paper.

Few implementations of Nussbaumer's algorithm exist. GMP [13] and MPIR [14] both have implementations available for multiplying large integers, libzn-poly [15] has an available implementation, and BCNS implemented it for their RLWE key exchange algorithm [16]. However, each of these implementations are non-vectorized and lack some of the optimizations of Nussbaumer's paper.

One more publicly available implementation exists, written for the paper 'Sieving for Shortest Vectors in Ideal Lattices: a Practical Perspective' [17]. This implementation [18], while still not implementing the optimizations of Nussbaumer, has been implemented to use vectorization through SSE instructions. However, not only does it implement a different set of parameters and does it lack vectorization of the forward transform, but most significantly it lacks the comparison with another negacyclic convolution algorithm on which we focus in this thesis.

Chapter 2

Preliminaries

The following describes some of the preliminaries for this thesis. Additionally, the reader is assumed to be familiar with Fourier transforms, in particular the decimation in time (DIT) fast Fourier transform, which is not covered separately here due to its wide usage. We refer readers unfamiliar with this subject to 'Inside the FFT Black Box' [19].

2.1 Convolutions

A core concept we consider in this thesis is that of convolutions.

Let $(a_0, a_1, \ldots, a_{n-1})$ and $(b_0, b_1, \ldots, b_{n-1})$ be sequences of length n. Then the acyclic convolution of these sequence is itself a length $(2n-1)$ sequence $(\tilde{z}_0, \tilde{z}_1, \ldots, \tilde{z}_{2n-2})$ with

$$\tilde{z}_i = \sum_{n+m=i} a_n b_m.$$

The cyclic convolution of $(a_i)_{i=0}^{n-1}$ and $(b_i)_{i=0}^{n-1}$ is then defined as a length n sequence $(z_i)_{i=0}^{n-1}$ with $z_i = \tilde{z}_i + \tilde{z}_{i+n}$, whereas the negacyclic convolution is defined as another length n sequence $(y_i)_{i=0}^{n-1}$ with $y_i = \tilde{z}_i - \tilde{z}_{i+n}$, where $(\tilde{z}_i)_{i=0}^{2n-2}$ is defined as above, padded with $\tilde{z}_{2n-1} = 0$.

These concepts have a strong correlation to that of modular polynomial multiplication. Let $A(u)$ and $B(u)$ be polynomials of degree at most $n-1$, with $A(u) = \sum_{i=0}^{n-1} a_i u^i$ and $B(u) = \sum_{i=0}^{n-1} b_i u^i$. Then it is easy to check that for $(\tilde{z}_i)_{i=0}^{2n-2}$ the acyclic convolution of $(a_i)_{i=0}^{n-1}$ and $(b_i)_{i=0}^{n-1}$, the following holds for the product of A and B:

$$A(u)B(u) = \sum_{i=0}^{2n-2} \tilde{z}_i u^i.$$

Note that for such polynomials A and B there exists exactly one polynomial $Z(u) \equiv A(u)B(u) \pmod{u^n - 1}$ and one polynomial $Y(u) \equiv A(u)B(u)$

(mod $u^n + 1$) for which the degrees of Z and Y are less than n. For these polynomials $Z(u) = \sum_{i=0}^{n-1} z_i u^i$ and $Y(u) = \sum_{i=0}^{n-1} y_i u^i$, it is easy to verify that $(z_i)_{i=0}^{n-1}$ and $(y_i)_{i=0}^{n-1}$ are exactly the cyclic convolution and the negacyclic convolution, respectively, of $(a_i)_{i=0}^{n-1}$ and $(b_i)_{i=0}^{n-1}$.

A core theorem on which Nussbaumer's negacyclic convolution algorithm is based is a discrete version of the cyclic convolution theorem as given in Theorem 1 in Appendix A, which states that the Fourier transform of a cyclic convolution of two sequences is equivalent to the pointwise product of the Fourier transform of these sequences.

2.2 Karatsuba's Algorithm

The Karatsuba algorithm is an algorithm that calculates the product of two polynomials. Here, we will cover a specialized version of Karatsuba that requires the input polynomials to have $2^n - 1$ coefficients for some n: this is true for the polynomials we consider in this thesis[1].

Let $p(u), q(u)$ be two such polynomials. We first split the polynomials in half, that is, we find $p_1(u), p_2(u), q_1(u), q_2(u)$ polynomials of degree at most $2^{n-1} - 1$ such that $p(u) = p_1(u) + p_2(u)u^{(2^{n-1})}$ and similar for $q(u)$. Then:

$$\begin{aligned} pq &= (p_1 + p_2 u^{(2^{n-1})})(q_1 + q_2 u^{(2^{n-1})}) \\ &= p_1 q_1 + (p_1 q_2 + p_2 q_1) u^{(2^{n-1})} + p_2 q_2 u^{2^n} \\ &= p_1 q_1 + [(p_1 + p_2)(q_1 + q_2) - p_1 q_1 - p_2 q_2] u^{(2^{n-1})} + p_2 q_2 u^{2^n} \end{aligned}$$

Karatsuba's algorithm exploits the realization that this latter form requires not four recursive polynomial multiplications, but three: one for $p_1 q_1$, one for $p_2 q_2$ and finally one for $(p_1 + p_2)(q_1 + q_2)$.

The number of steps these calculations require is now $T(n) = 3T(\frac{n}{2}) + O(n)$. The complexity of this algorithm follows from the master theorem as $\Theta(n^{\log_2(3)})$, while the schoolbook algorithm with one extra polynomial multiplication would have complexity $\Theta(n^2)$.

When performing the algorithm as specified, some additions turn out to be executed more than once. Figure 2.1 shows a schematic overview of the additions. The lower half of the result requires the calculation of the upper half of $p_1 q_1$ minus the lower half of $p_2 q_2$. However, the upper half of the result requires the calculation of the lower half of $p_2 q_2$ minus the upper half of $p_1 q_1$: note that the two are sign inversions of each other, so the latter can be calculated by subtracting the first, omitting these additions.

Throughout this thesis we do not directly use the output of Karatsuba's algorithm, but first reduce modulo $u^n + 1$ to calculate a negacyclic convolution. This can be calculated by subtracting the upper half of the output

[1] Of course one could append coefficients with value 0 in the general case, but this would generally not be the most efficient method of applying Karatsuba's algorithm.

from the lower half. It turns out that doing this we again duplicate the calculation of certain additions, in this case the adding of the lower half of p_1q_1 and the upper half of p_2q_2, which can similarly be omitted.

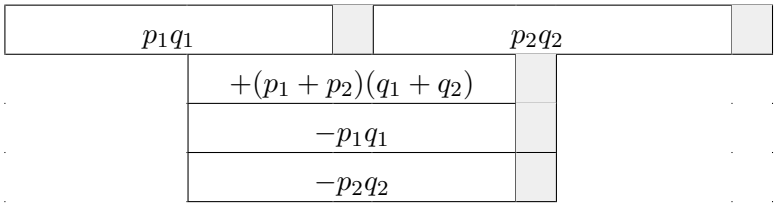

Figure 2.1: Schematic overview of additions/subtractions required for Karatsuba's algorithm, where light-gray fields indicate a single coefficient that is always 0, due to the fact that the given polynomial can be of degree of at most $2^n - 2$.

Chapter 3

Nussbaumer's Negacyclic Convolution Algorithm

In his paper Nussbaumer [9] proposes an algorithm for calculating the negacyclic convolution of two N-sequences, where N is a power of two, say $N = 2^n$. Shifting perspective to that of polynomials with indeterminate u, this equates to the calculation of the product, reduced modulo $u^N + 1$, of two polynomials; that is, the result being the remainder from dividing the product by $u^N + 1$. This chapter discusses Nussbaumer's algorithm.

3.1 Overview

Nussbaumer's algorithm is based on the observation in Theorem 2 (see Appendix A), being that for a given ring $\mathbf{R}$, and for $v_1, v_2 \in \mathbb{N}$ with $v_1 \leq v_2$ and $v_1 + v_2 = n$, and defining $m = 2^{v_1}$ and $r = 2^{v_2}$ (and hence $rm = N$ and $m|r$), that

$$\mathbf{R}[u]/(u^N + 1) \cong (\mathbf{R}[u_1]/(u_1^r + 1))[u_2]/(u_2^m - u_1)$$

by the map (which we will call Ψ)

$$\sum_{i=0}^{N-1} a_i u^i \mapsto \sum_{i=0}^{m-1} \sum_{j=0}^{r-1} a_{mj+i} u_1^j u_2^i.$$

As the resulting negacyclic convolution Z is reduced modulo $u^N + 1$, its degree can never exceed $N-1$, hence we can write $Z = \sum_{i=0}^{N-1} z_i u^i$, where the coefficients $(z_i)_{i=0}^{N-1}$ are to be calculated. Accordingly, given $X = \sum_{i=0}^{N-1} x_i u^i$ and $Y = \sum_{i=0}^{N-1} y_i u^i$, with $X, Y \in \mathbf{R}[u]/(u^N + 1)$, we find such Z with $Z = XY$.

From the isomorphism property it follows that

$$Z = \Psi^{-1}(\Psi(X)\Psi(Y)),$$

where as per definition, if we set $X_i = \sum_{j=0}^{r-1} x_{mj+i} u_1^j$ and similar for Y_i,

$$\Psi(X)\Psi(Y) = \left(\sum_{i=0}^{m-1} X_i u_2^i\right)\left(\sum_{i=0}^{m-1} Y_i u_2^i\right).$$

Note that $\left(\sum_{i=0}^{m-1} X_i u_2^i\right)\left(\sum_{i=0}^{m-1} Y_i u_2^i\right)$ is a polynomial with degree at most $2m-2$, so reduction modulo $u_2^{2m}-1$ has no effect. Consequently the product $\Psi(X)\Psi(Y)$ can be calculated as the length $2m$ cyclic convolution of $(X_i)_{i=0}^{2m-1}$ and $(Y_i)_{i=0}^{2m-1}$ with X_i and Y_i set to 0 for $i \geq m$, as the cyclic, negacyclic and acyclic convolution each produce the same result.

In order to calculate this cyclic convolution, note that $\omega = u_1^{r/m}$ is a $2m$-th principal root of unity in $\mathbf{R}[u_1]/(u_1^r+1)$ as shown in Theorem 3 (Appendix A), hence according to Theorem 1 the Fourier transform of this convolution can be calculated as the pointwise product of the Fourier transform of the $2m$-sequences. These transforms can be calculated efficiently with a fast Fourier transform algorithm, exploiting the fact that since the calculations occur modulo u_1^r+1, multiplication by ω can be performed merely by rotating coefficients and inverting the sign of overflows.

After performing the pointwise multiplications, themselves being negacyclic convolutions of length r due to the ring the coefficients reside in, followed by the inverse fast Fourier transform of the result we find the cyclic convolution $(D_i)_{i=0}^{2m-1}$ of $(X_i)_{i=0}^{2m-1}$ and $(Y_i)_{i=0}^{2m-1}$. As such, $\Psi(Z) \equiv \sum_{i=0}^{2m-1} D_i u_2^i \pmod{u_2^m - u_1}$.

The final step before trivially being able to perform the inverse Ψ^{-1} is to find $(Z_i)_{i=0}^{m-1}$ such that

$$\sum_{i=0}^{2m-1} D_i u_2^i \equiv \sum_{i=0}^{m-1} Z_i u_2^i \pmod{u_2^m - u_1}.$$

Note that

$$\begin{aligned}
\sum_{i=0}^{2m-1} D_i u_2^i &\equiv \sum_{i=0}^{m-1} D_i u_2^i + \sum_{i=m}^{2m-1} D_i u_2^i \\
&\equiv \sum_{i=0}^{m-1} D_i u_2^i + \sum_{i=m}^{2m-1} D_i u_1 u_2^{i-m} \pmod{u_2^m - u_1} \\
&\equiv \sum_{i=0}^{m-1} D_i u_2^i + \sum_{i=0}^{m-1} u_1 D_{i+m} u_2^i \\
&\equiv \sum_{i=0}^{m-1} [D_i + u_1 D_{i+m}] u_2^i \pmod{u_2^m - u_1},
\end{aligned}$$

hence we get $Z_i = D_i + u_1 D_{i+m}$, noting again that multiplication by u_1 only requires rotations of coefficients and sign inversion of overflows. The

final step of performing Ψ^{-1} is then trivial performed as a shuffle (more specifically a transpose) of coefficients.

This shows that the negacyclic convolution can be calculated as follows (the case for $N = 2$ being excluded as a simple base case):

1. Pick m and r under above conditions.
2. Set $X_i = \sum_{j=0}^{r-1} x_{mj+i} u_1^j$ and $Y_i = \sum_{j=0}^{r-1} y_{mj+i} u_1^j$ for $0 \leq i < m$ and $X_i = Y_i = 0$ for $m \leq i < 2m$.
3. Perform a fast Fourier transform with $2m$-th principal root $\omega = u_1^{r/m}$ on $(X_i)_{i=0}^{2m-1}$ and $(Y_i)_{i=0}^{2m-1}$; name these $(\tilde{X}_i)_{i=0}^{2m-1}$ and $(\tilde{Y}_i)_{i=0}^{2m-1}$, respectively.
4. Calculate $(\tilde{Z}_i)_{i=0}^{2m-1}$ as the pointwise product of $(\tilde{X}_i)_{i=0}^{2m-1}$ and $(\tilde{Y}_i)_{i=0}^{2m-1}$; more specifically $\tilde{Z}_i$ is the negacyclic convolution of $\tilde{X}_i$ and $\tilde{Y}_i$. This can be done through recursion or some alternative algorithm.
5. Perform an inverse fast Fourier transform of $(\tilde{Z}_i)_{i=0}^{2m-1}$; call this $(D_i)_{i=0}^{2m-1}$ (note that $2m \in \mathbf{R}$ needs to be invertible for this step).
6. Say $D_i = \sum_{j=0}^{r-1} d_{ij} u_1^j$. Find $(z_i)_{i=0}^{n-1}$ with $Z_i = \sum_{j=0}^{r-1} z_{mj+i} u_1^j$, such that $Z_i \equiv D_i + u_1 D_{i+m}$.

Then $(z_i)_{i=0}^{n-1}$ is the negacyclic convolution of $(x_i)_{i=0}^{n-1}$ and $(y_i)_{i=0}^{n-1}$.

3.2 Parameter Choice

The algorithm described thus far still has a degree of freedom, namely the choice of r or m (the other is then defined through $mr = N$). The optimal choice of this parameter clearly depends on the algorithm used to calculate the length r negacyclic convolution of Step 4 in the previous section.

In his paper Nussbaumer suggests that "it is often advantageous to use only one small polynomial product algorithm to construct the various polynomial products required for the cyclic convolution of length N. In particular, choosing $(u^2 + 1)$ as the basic building block with which all other polynomial products are constructed is particularly interesting, since only one algorithm with 3 multiplications and 3 additions need to be stored." [9][1]. As such, both Knuth's implementation and Nussbaumer's concept recurse into the algorithm itself, and both algorithms use $r = 2^{\lceil n/2 \rceil}$ for each instance.

As we will see in Section 5.6.1, recursing into Nussbaumer's algorithm is not optimal for our optimized implementation (and most implementations in

[1] Notation in this citation has been adapted to ours. Also note that as explained later, the number of additions required is different if all transforms are considered towards the operation count, which we will assume.

general). Instead, we will use Karatsuba's algorithm to calculate the smaller negacyclic convolutions.

In Chapter 4 we consider the optimizations Nussbaumer proposes in his paper. In order to analyze these optimizations as envisioned by Nussbaumer we will imitate the method Nussbaumer proposed; hence in that chapter we will recurse into Nussbaumer's algorithm with the same r as proposed in his paper. However, as we will not use this method for the final implementation, arguing the choice of r in Nussbaumer's application is outside the scope of this thesis.

3.3 Implementation

Nussbaumer's algorithm is made explicit in the form of a well-defined list of calculations[2] by Knuth [10]. We base our initial, unoptimized code on this version; as do the implementations publicly available. In this section the corresponding pseudo-code is given and its correctness is argued by showing the correlation with the steps provided in the mathematical description of Section 3.1. In the code the naming conventions of Knuth have been preserved, but the code has been changed slightly with respect to Knuth's description for sake of clarity.

The algorithm is subdivided into three parts. First, we show the algorithm for calculating the forward transforms, similar to Steps 2 and 3 of the description in Section 3.1. Second, we show the inverse transform from Steps 5 and 6 in the same section. Finally, we wrap this into the full algorithm that uses the other two in order to implement all steps.

Throughout this chapter we take the notation $\mathbf{BitRev}_n(a)$ to be the reverse of the lower n bits of an index a, and $X(u)$ **mod** $Y(u)$ to be the remainder of dividing the polynomial $X(u)$ by the polynomial $Y(u)$.

3.3.1 Forward Transform

First we introduce the forward transform similar to Steps 2 and 3 of Section 3.1. The result is not identical: we calculate $\tilde{X}$ and $\tilde{Y}$ as the fast Fourier transform where the output is in bit-reversed order, as this allows for a somewhat simpler implementation [19]. It will be shown later that this is acceptable. The pseudo-code for this part of the algorithm is given in Algorithm 1.

The butterfly operations in lines 16 and 17 can efficiently be calculated by combining the multiplication (which is merely rotating and sign inversions) with the addition in order to prevent the sign inversions to be performed

[2] It is too high-level to be referred to as pseudo-code, but lower level and more explicit than the mathematical description given in Section 3.1.

Algorithm 1 Knuth's version of Nussbaumer's negacyclic convolution transformation algorithm.

Precondition: Let $n, v_1, v_2 \in \mathbb{N}$, $N = 2^n$, $m = 2^{v_1}$ and $r = 2^{v_2}$ with $mr = N$ and $m|r$. Let $\mathbf{R}$ be a ring, and let $X = (x_i \in \mathbf{R})_{i=0}^{N-1}$ be a length N sequence.

Postcondition: Returns the transformed polynomial as described in Nussbaumer's algorithm Step 2 and 3 of Section 3.1, with the output in bit-reversed order.

function TRANSFORM(N, m, r, X)
    ▷ *Initialize $\tilde{X}$ for the fast Fourier transform.*
    for $i \leftarrow 1$ **to** $m - 1$ **do**
        for $j \leftarrow 1$ **to** $r - 1$ **do**
            $\tilde{X}_{ij} \leftarrow x_{mj+i}$
            $\tilde{X}_{(i+m)j} \leftarrow x_{mj+i}$

    ▷ *Perform all but the first step in the fast Fourier transform.*
    ▷ *Here $\tilde{X}_i(u)$ indicates the polynomial $\sum_{j=0}^{r-1} \tilde{X}_{ij} u^j$.*
    for $j \leftarrow \lfloor n/2 \rfloor - 1$ **to** 0 **do**
        for $\hat{s} \leftarrow 0$ **to** $m/2^j - 1$ **do**
            $s \leftarrow \hat{s} \cdot 2^{j+1}$
            $s' \leftarrow \mathbf{BitRev}_{\lfloor n/2 \rfloor - j}(\hat{s}) \cdot 2^j$
            for $t \leftarrow 0$ **to** $2^j - 1$ **do**
                ▷ *Perform the Cooley-Tukey butterfly [19]*
                $Z(u) \leftarrow \tilde{X}_{s+t}(u) - \{u^{s'r/m} \tilde{X}_{s+t+2^j}(u) \bmod (u^r + 1)\}$
                $\tilde{X}_{s+t}(u) \leftarrow \tilde{X}_{s+t}(u) + \{u^{s'r/m} \tilde{X}_{s+t+2^j}(u) \bmod (u^r + 1)\}$
                $\tilde{X}_{s+t+2^j}(u) \leftarrow Z(u)$
    return $\tilde{X}$

explicitly. These lines can thus be implemented merely with additions and subtractions.

Note the similarity between the second part of the algorithm (lines 10 to 18) and the DIT FFT algorithm with natural ordered input and bit-reverse ordered output. Should the iteration of the outer loop start with the value $j = \lfloor n/2 \rfloor$ rather than $\lfloor n/2 \rfloor - 1$, then it would be a regular implementation of said algorithm (cf. algorithm 7.2 of 'Inside the FFT Black Box' [19], where the variables *Distance*, K and J in the book are represented by 2^j, $\hat{s}$ and $s + t$ in this algorithm, respectively).

This apparent discrepancy between the DIT FFT with bit-reverse ordered output and this part of the algorithm stems from an optimization exploiting the property of the input $(X_i)_{i=0}^{2m-1}$ that the second half is set to 0. The result of an iteration with $j = \lfloor n/2 \rfloor$ is then easily confirmed to be the first half repeated twice. This optimizes the calculations as the first $2m$ polynomial additions of the FFT algorithm need not be calculated.

The first half of the algorithm, lines 3 to 6, initializes the input for the second half in this manner. It reorders the coefficients to provide the X_i (respectively Y_i) in intuitive order (as per Step 2 of Section 3.1) and duplicates the output twice to mirror the first iteration of the fast Fourier transform.

It follows that Algorithm 1 performs Steps 2 and 3 of Section 3.1, with the exception that the output is in bit-reversed order; that is, where the order of the output is such that a coefficient is positioned at the index with its binary representation mirrored with respect to the index of its position in the order of output of a regular Fourier transform. For details, see 'Inside the FFT Black Box' [19].

3.3.2 Inverse Transform

The second part of the algorithm as described by Knuth is the inverse transform as described in Steps 5 and 6 in Section 3.1. Similar to the reasoning for the forward transform the input is provided in bit-reversed order; this furthermore allows the output of the former to be used as input of the latter without a need for reordering.

The pseudo-code for this algorithm is shown in Algorithm 2. Note again that the multiplication by $u^{-s'r/m}$ modulo $u^r + 1$ can be calculated through rotating and sign inversion of coefficients.

We can see that the first part of the algorithm, lines 3 to 12, performs the inverse of the fast Fourier transform of Algorithm 1. Namely, j iterates though the same values a standard forward fast Fourier transform would, but in reverse order, where each iteration updates $\tilde{Z}_i$ for $i \in \{0, 1, \ldots, 2m-1\}$ exactly once, for the same pairs $\tilde{X}_{s+t}$ and $\tilde{X}_{s+t+2^j}$ as in their corresponding iterations in the forward transform.

Algorithm 2 Knuth's version of Nussbaumer's negacyclic convolution inverse transformation algorithm.

Precondition: Let $n, v_1, v_2 \in \mathbb{N}$, $N = 2^n$, $m = 2^{v_1}$ and $r = 2^{v_2}$ with $mr = N$ and $m|r$. Let $\mathbf{R}$ be a ring where $2 \in \mathbf{R}$ is invertible. Let $\tilde{Z} = (\tilde{Z}_i)_{i=0}^{2m-1}$ be a length $2m$ sequence of length r sequences in $\mathbf{R}$.

Postcondition: Returns the coefficients of the polynomial after inverse transformation as described in Nussbaumer's algorithm Steps 5 and 6 of Section 3.1, where the input is in bit-reversed order.

1: **function** InverseTransform($N, m, r, \tilde{Z}$)
2: ▷ *Perform the full inverse Fourier transform.*
3: **for** $j \leftarrow 0$ **to** $\lfloor n/2 \rfloor$ **do**
4: **for** $\hat{s} \leftarrow 0$ **to** $m/2^j - 1$ **do**
5: $s \leftarrow \hat{s} \cdot 2^{j+1}$
6: $s' \leftarrow \mathbf{BitRev}_{\lfloor n/2 \rfloor - j}(\hat{s}) \cdot 2^j$
7: **for** $t \leftarrow 0$ **to** $2^j - 1$ **do**
8: $i_1 \leftarrow s + t$
9: $i_2 \leftarrow s + t + 2^j$
10: $Y(u) \leftarrow \frac{1}{2}(\tilde{Z}_{i_1}(u) + \tilde{Z}_{i_2}(u))$
11: $\tilde{Z}_{i_2}(u) \leftarrow \frac{1}{2}u^{-s'r/m}(\tilde{Z}_{i_1}(u) - \tilde{Z}_{i_2}(u)) \bmod (u^r + 1)$
12: $\tilde{Z}_{i_1} \leftarrow Y(u)$
13:
14: ▷ *Perform Step 6 of the algorithm described in Section 3.1*
15: **for** $i \leftarrow 0$ **to** $m - 1$ **do**
16: $z_i \leftarrow \tilde{Z}_{i0} - \tilde{Z}_{(m+i)(r-1)}$
17: **for** $j \leftarrow 0$ **to** $r - 1$ **do**
18: $z_{mj+i} \leftarrow \tilde{Z}_{ij} + \tilde{Z}_{(m+i)(j-1)}$
19: **return** $(z_i)_{i=0}^{2m-1}$

These steps in the forward transform calculate (albeit in-place):

$$\tilde{X}'_{s+t}(u) \leftarrow \tilde{X}_{s+t}(u) + \{u^{s'r/m}\tilde{X}_{s+t+2^j}(u) \ (\textbf{mod } u^r + 1)\}$$

$$\tilde{X}'_{s+t+2^j}(u) \leftarrow \tilde{X}_{s+t}(u) - \{u^{s'r/m}\tilde{X}_{s+t+2^j}(u) \ (\textbf{mod } u^r + 1)\}$$

For which we can easily see that

$$\tilde{X}'_{s+t}(u) + \tilde{X}'_{s+t+2^j}(u) = 2\tilde{X}_{s+t}(u), \text{ and}$$

$$\tilde{X}'_{s+t}(u) - \tilde{X}'_{s+t+2^j}(u) \equiv 2u^{s'r/m}\tilde{X}_{s+t+2^j}(u) \bmod (u^r + 1).$$

It follows that lines 10 to 12 perform the inverse of the Cooley-Tukey butterfly, and thus that this part calculates the inverse of the Fourier transform (with input in bit-reversed order). Note that this deviates from a regular inverse Fourier transform as the constant multiplications would normally be accumulated to after the final round of butterflies.

The second part of the algorithm, lines 15 to 18 sets z_{mj+i} to the j-th component of the remainder of dividing $\tilde{Z}_i(u) + u\tilde{Z}_{m+i}(u)$ by $u^r + 1$ (whose calculation is made explicit), which matches Step 6 in the algorithm of Section 3.1.

3.3.3 Negacyclic Convolution

The algorithm combining these two components is shown in Algorithm 3. The core of the algorithm, lines 8 to 14, used when $N \neq 2$, transforms the two polynomials, performs pointwise negacyclic convolutions, and finally uses the inverse transform. This is in accordance to the steps given in Section 3.1. In order to calculate the $2m$ length r negacyclic convolutions the algorithm recurses into itself; it is noteworthy that other methods may be used instead in actual implementations.

We use $N = 2$ as a base case for which, as Nussbaumer's algorithm is no longer applicable lacking a sane choice for r and m, an alternative algorithm is applied as seen in lines 3 to 6. It is easy to verify that these lines indeed perform the negacyclic convolution for this case. It does so using three multiplications and five additions[3], unlike the schoolbook method which uses four multiplications and two additions. As a multiplication operation is significantly more expensive than an addition on most modern processors, including Intel and AMD according to the instruction tables by Agner [20], the version used by Knuth generally requires fewer clock cycles.

[3]Nussbaumer's paper assumes one of the input polynomials is fixed and does not count operations towards the preparative calculations that can be done on it. As such, only 3 additions are considered to be required, as $x_0 + x_1$ and $x_1 - x_0$ can be calculated for free. In our application no polynomial is fixed, so we will include these operations in the total count.

In the algorithm specified here the values for m and r are assigned to be $2^{\lfloor n/2 \rfloor}$ and $2^{\lceil n/2 \rceil}$, respectively. This is not a requirement for Nussbaumer's algorithm, but it is explicitly set as such in Knuth's implementation. Furthermore, it is the method Nussbaumer suggests and assumes in his analysis, so we use the same for a fair comparison.

We remind the reader that the output of both forward transforms is in bit-reversed order. As the pointwise negacyclic convolutions are independent of the ordering, the input for the inverse transform maintains this. Indeed, the inputs for the inverse transform is also bit-reversed, as expected by the given implementation in the previous section. As such the lack of re-ordering has no effect on the result while simplifying both the forward and inverse fast Fourier transform algorithms.

Algorithm 3 Knuth's version of Nussbaumer's negacyclic convolution algorithm.

Precondition: Let $n \in \mathbb{N}$, $N = 2^n$. Let $\mathbf{R}$ be a ring where $2 \in \mathbf{R}$ is invertible. Let $X = (x_i \in \mathbf{R})_{i=0}^{N-1}$ and $Y = (y_i \in \mathbf{R})_{i=0}^{N-1}$ be two length N sequences.

Postcondition: Returns the negacyclic convolution of X and Y.

1: **function** NUSSBAUMER(N, X, Y)
2: **if** N = 2 **then** ▷ Base case for recursion
3: $t \leftarrow x_0(y_0 + y_1)$
4: $z_0 \leftarrow t - (x_0 + x_1)y_1$
5: $z_1 \leftarrow t + (x_1 - x_0)y_0$
6: **return** $(z_i)_{i=0}^{N-1}$
7:
8: $m \leftarrow 2^{\lfloor n/2 \rfloor}$
9: $r \leftarrow 2^{\lceil n/2 \rceil}$
10: $\tilde{X} \leftarrow \text{Transform}(N, m, r, X)$
11: $\tilde{Y} \leftarrow \text{Transform}(N, m, r, Y)$
12: **for** $i \leftarrow 0$ **to** $2m - 1$ **do**
13: $\tilde{Z}_i \leftarrow \text{Nussbaumer}(r, \tilde{X}, \tilde{Y})$ ▷ Where $\tilde{Z}_i = (\tilde{Z}_{ij})_{j=0}^{r-1}$
14: **return** InverseTransform($N, \tilde{Z}$)

3.4 Results

In the referenced source code the test for this implementation of the algorithm is named 'Knuth version'. As ring, $\mathbf{R} = \mathbb{Z}/12289\mathbb{Z}$ is used as in accordance with the choice of q in the New Hope paper [8] and for the same reasons we set $N = 1024$.

	Additions	Mults	Const Mults
Transformation	10 240	0	0
Recursion	186 624	24 576	61 440
Inverse transformation	15 376	0	12 288

Table 3.1: Number of operations for Knuth's description of Nussbaumer's negacyclic convolution algorithm. 'Const Mults' lists the multiplications by constants, which are omitted in the 'Mults' column.

The test calculates the negacyclic convolution of two random polynomials (where the random polynomial generation routines of New Hope are used) through both this implementation of Nussbaumer's algorithm and through an implementation of the schoolbook algorithm; the latter is used to confirm the results of the former.

Note that the source code is written to be easy to follow, but completely unoptimized. In the next chapter we will reduce the number of operations for this implementation, followed in the subsequent chapter with the presentation of an optimized and vectorized form of Nussbaumer's algorithm.

Table 3.1 shows the number of operations required to calculate the negacyclic convolution by the given implementation of Knuth's description of Nussbaumer's algorithm. In the table we count sign inversions as additions.

Chapter 4

Nussbaumer's Optimizations

The algorithm described and implemented so far merely encompasses a part of Nussbaumer's observations; it is a version consisting of only his general ideas and observations, and is in the form as given by Knuth [10]. However, in his paper Nussbaumer proposes several optimizations to decrease the number of ring operations required.

This chapter will describe such optimizations by Nussbaumer, along with a few trivial optimizations in the number of ring operations that can be made from Knuth's version that are not explicitly stated in Nussbaumer's paper.

A significant note on this matter by Nussbaumer is that "one of the input sequences [...] is usually fixed" [9]. If it is indeed fixed, then one forward transform and some preparative steps for the pointwise multiplications that only depend on this input can be considered free, as is assumed by Nussbaumer.

As we will see in the following chapter this is not generally the case in the New Hope algorithm, where most polynomials are distinct for each key exchange; with the exception of two transforms that could be re-used exactly once. Only for one of the input sequences, $\mathbf{a}$, it is stated that "if in practice it turns out to be too expensive to generate $\mathbf{a}$ for every connection, it is also possible to cache $\mathbf{a}$ on the server side for, say a few hours without significantly weakening the protection against all-for-the-price-of-one attacks" [8].

While this would suggest one input sequence can be transformed almost for free (at least being amortized for these few hours), we shall not depend on this potential optimization as it poses constraints on the usage of the algorithm which has a potential effect on the algorithm's security. As such we shall assume a general case where $\mathbf{a}$ is not cached, and as such we will not consider one of the transforms to be free like Nussbaumer does in his paper.

This explains why some optimizations we propose here are not considered by Nussbaumer: it only reduces some operations that he would consider to

be free. As they are not considered free in our context, they are counted as optimizations here.

Another deviation of our counting of operations with respect to Nussbaumer's is that we consider sign inversions to be additions. They may be implemented more efficiently however, as reductions can be mitigated. Nussbaumer does not count these sign inversions to the operation count at all, as can be seen due to the fact that Step 6 in appendix A in his paper is considered free, even though sign inversions are clearly needed. In this thesis we count sign inversions to be equal to additions and subtractions. This, too, causes us to consider optimizations that are irrelevant with respect to Nussbaumer's method of operation counting.

The names of the optimization in the results table correspond to their name in the provided source code, given in the directory 'tests'.

4.1 Multiplication Factor Accumulation

We first consider Algorithm 2, lines 3 to 12. Here, after each iteration of j every new value of $\tilde{Z}_i(u)$ for $i \in \{0, 1, \ldots, 2m\}$ is the sum or difference of two elements from the end of the previous iteration, one of which multiplied by some value (depending on j and i), with the total again multiplied by $\frac{1}{2}$.

As each step forms linear combinations with each value having an equal number of multiplications by $\frac{1}{2}$, it is clear that rather than multiplying with $\frac{1}{2}$ at the end of each iteration we can accumulate these multiplications and perform this after the complete transform. By pre-calculating the resulting factor of $(\frac{1}{2})^{\lfloor n/2 \rfloor + 1} = \frac{1}{2m}$ and multiplying it in one round, most of these multiplications can thus be mitigated. Note that after this optimization we end up with an implementation of a standard inverse Fourier transform as we will show in the next section.

The results are shown in Table 4.1, listed as 'Knuth optimized 1' corresponding to its name in the associated source code. The number of constant multiplications of the recursion and inverse transformation steps have decreased by respectively 53% and 83%.

4.2 Inverse Transform Modification

After removing the multiplication factor as described in the previous optimization, this portion of the algorithm becomes identical to a DIF FFT with bit-reversed ordered input with $u^{-r/m}$ as $2m$-th root of unity (cf. algorithm 5.2 in 'Inside the FFT Black Box' [19], where *Distance* (= *NumOfProblems*), *JFirst* and J take the place of 2^j, t and $s + t$, respectively). Lines 10 to 12 form a Gentleman-Sande butterfly. The result of the fast Fourier transform is then multiplied by the accumulated constant found in the previous section.

		Transf	Recurse	Transf^{-1}	Correct
Knuth version	Adds	10 240	186 624	15 376	
	Mults	0	24 576	0	
	CMults	0	61 440	12 288	
Knuth optimized 1	Adds	10 240	186 624	15 376	
	Mults	0	24 576	0	
	CMults	0	28 672 (-32 768)	2 048 (-10 240)	
Knuth optimized 2	Adds	10 240	182 272 (-4 352)	13 312 (-2 064)	
	Mults	0	24 576	0	
	CMults	0	28 672	2 048	
Knuth optimized 3	Adds	10 208 (-32)	126 567 (-55 705)	15 264 (+1 952)	
	Mults	0	11 907 (-12 669)	0	
	CMults	0	21 672 (-7 000)	2 048	
Knuth optimized 4	Adds	10 208	126 567	15 264	
	Mults	0	11 907	0	
	CMults	0	0 (-21 672)	0 (-2 048)	1 024 (+1 024)

Table 4.1: Number of additions, multiplications by constants (CMults) and non-constant multiplications (Mults) for both the original implementation of Knuth's version of the Nussbaumer algorithm and the optimizations discussed in Chapter 4 of the forward transformation, recursion and inverse transformation steps (with in parenthesis the difference between this and the previous version, if any).

This version of the FFT algorithm has a downside for our implementation. Namely, it multiplies by $u^{-s'r/m}$ (modulo $u^r + 1$), which requires sign inversions[1], rather than the DIT FFT algorithm which combines the multiplication with additions or subtractions, which can be combined to avoid the need of explicit sign inversions.

While this can be solved by changing the implementation of the DIF FFT (by not performing the rotation until the next iteration for j, noting that the last iteration does not require rotations at all), this results in an awkward dependency between different iterations. Rather than crafting an ugly fix for our algorithm, we move to the commonly used DIT FFT algorithm, which as stated before easily and cleanly solves the problem.

The results are shown in Table 4.1, listed as 'Knuth optimized 2'. The number of additions of the recursion and inverse transformation steps are decreased by 2% and 13%, respectively.

4.3 Removal of Convolution

One observation by Nussbaumer allows for one recursive call calculating the negacyclic convolution of length r sequences to be omitted. This is based on the observation that as $(X_i)_{i=0}^{2m-1}$ and $(Y_i)_{i=0}^{2m-1}$ both have their upper half set to 0, their cyclic convolution $(D_i)_{i=0}^{2m-1}$ has $D_{2m-1} = 0$. This can easily be seen shifting perspective to that of polynomials: the product of two polynomials of degree at most $m-1$ itself has a degree of at most $2m-2$.

Hence, for $i = 0, 1, \ldots, 2m-1$ we have

$$D_i = D_i - D_{2m-1}.$$

Using the notation from the previous chapter, where $(\tilde{Z}_i)_{i=0}^{2m-1}$ is the input into the inverse Fourier transform (that is, the pointwise product of the transformed inputs) with $2m$-th root of unity $\omega = {u_1}^{-r/m}$, the result of which being $(D_i)_{i=0}^{2m-1}$, it follows by definition of the inverse Fourier transform that

$$2^{\lfloor n/2 \rfloor + 1} D_i = \sum_{k=0}^{2m-1} \tilde{Z}_k(u_1)\omega^{ik}.$$

[1]Note again that in his article, Nussbaumer does not consider sign inversions as additions, as we do.

So

$$\begin{aligned} 2^{\lfloor n/2 \rfloor+1} D_i &= 2^{\lfloor n/2 \rfloor+1} D_i - 2^{\lfloor n/2 \rfloor+1} D_{2m-1} \\ &= \sum_{k=0}^{2m-1} \tilde{Z}_k(u_1)\omega^{ik} - \sum_{k=0}^{2m-1} \tilde{Z}_k(u_1)\omega^{(2m-1)k} \\ &= \sum_{k=0}^{2m-1} \tilde{Z}_k(u_1)(\omega^{ik} - \omega^{(2m-1)k}). \end{aligned}$$

Since $\omega^{ik} - \omega^{(2m-1)k} = 0$ for $k = 0$ we obtain

$$2^{\lfloor n/2 \rfloor+1} D_i = \sum_{k=1}^{2m-1} \tilde{Z}_k(u_1)(\omega^{ik} - \omega^{(2m-1)k}).$$

As a result, when exploiting this fact the result of $\tilde{Z}_0$ is never used and as such we need not calculate it; we can set this to any value prior to calculating the inverse Fourier transform, as long as afterwards D_{2m-1} is subtracted from each D_i.

This comes at the expense of $2m-1$ polynomial subtractions (we need not subtract Z_{2m-1} from itself), so $2mr - r = 2N - r$ ring additions. However, we can omit some of these additional additions in both the forward and inverse Fourier transforms through two observations.

First, in the forward Fourier transformations, note that $\tilde{X}_0$ and $\tilde{Y}_0$ need not be calculated, so the polynomial additions in the final iteration of the forward Fourier transforms that calculate these are unnecessary. This decreases the number of additions required for the forward transformations by r per transform, so $2r$ in total.

Second, before the inverse Fourier transform we can choose to set $\tilde{Z}_0^0 = 0$, where $\tilde{Z}_i^n$ represents the value of $\tilde{Z}_i$ after n iterations for j in the Fourier transform. During the first iteration of j, we set

$$\tilde{Z}_0^1 = \tilde{Z}_0^0 + \tilde{Z}_1^0 = \tilde{Z}_1^0, \text{ and}$$

$$\tilde{Z}_1^1 = \tilde{Z}_0^0 - \tilde{Z}_1^0 = -\tilde{Z}_1^0.$$

So the polynomial addition for $\tilde{Z}_0^1$ need not be calculated, and the polynomial addition of $\tilde{Z}_1^1$ can be done only through sign inversion (which Nussbaumer does not count as addition, but we do).

Hence $3r$ extra additions can be mitigated, not counting those in the recursion calls. As such, implementing this optimization has a cost of $2N - 4r$ extra additions. On the other hand we gain the amount of operations required for the calculation of a single length r negacyclic convolution.

Whether this is indeed an optimization depends on the recursion method used, the method used for counting operations and the considered ratio of weights for multiplication to addition operations.

From Nussbaumer's perspective this modification would always become an optimization after performing the matrix exchange algorithm explained later, which will place the additional cost in a single forward transform that is considered to be free. In other contexts whether this is actually an optimization should be considered separately; we shall not do so for the current implementation as another recursion method will be used in the actual implementation of Chapter 5, making such results superfluous.

The operation counts after implementing this modification are shown in Table 4.1, listed as 'Knuth optimized 3'. Note that the number of multiplications for the recursion step has decreased by 52% for the non-constant multiplications and 24% for the constant multiplications, while the inverse transformation additions has increased by 15%. The number of additions in the recursion step decreases by 30%, while the number of additions in the forward transform has decreased slightly, by less than 1%.

4.4 Recursion Factor Accumulation

An observation Nussbaumer does not make allows us to decrease the number of multiplications further[2]. Namely we can remove the multiplication we created in Section 4.1 entirely from the inverse transform algorithm. As the steps of the inverse transform algorithm after the inverse Fourier transform are trivially linear the result of the algorithm will be the expected negacyclic convolution multiplied by $2m$.

Note that each recursive call to the Nussbaumer algorithm will then be off by a factor that depends only on the input value N (for $N = 2$ the output is correct as we fall back to another algorithm). As the inverse Fourier transform itself is also linear [21] this factor simply propagates through the inverse transform function. Each recursive step will add such a factor, which can be removed in a single sweep of multiplications of all N terms after the result has been calculated. This last step is called the 'Correction' step and used only once for each instantiation of the algorithm.

Not only does this correction step supersede the multiplications in the recursion calls; it also reduces the number of the multiplications by postponing it to after the last steps of the algorithm. Recall that the inverse transform has $2m$ polynomials as output, requiring $2mr = 2N$ multiplications for a regular inverse Fourier transform, whereas the output of the entire algorithm only has N coefficients that need to be multiplied.

The results are shown in Table 4.1, listed as 'Knuth optimized 4'. The multiplications of the inverse transform moved to the correction and were halved. The removal of the multiplications in the inverse transform during

[2]The fact Nussbaumer does not note this is because in the optimization presented in the next section these multiplications are moved to one of the initial transforms, which we remind the reader is considered to be free from his perspective.

the recursion calls results in fewer multiplications in the recursion step: all the constant multiplications were omitted in the recursion step.

4.5 Matrix Exchange Algorithm

In his paper, Nussbaumer mentions an additional optimization step that is advantageous if one polynomial transformation can be re-used multiple times; that is, if the same polynomial is multiplied multiple times. He achieves this by applying "a matrix exchange algorithm similar to that given in [22]" [9], which allows the more complex matrix responsible for performing the inverse transform to be exchanged with one of the cheaper forward transform matrices. In this section we cover the matrix exchange algorithm.

4.5.1 Exchange

In order to perform the exchange algorithm, we first write the calculations of Nussbaumer's algorithm in the codomain of the isomorphism Ψ (we shall refer to this as 'Nussbaumer's domain') in the form

$$z = C((Ax) \otimes (By)).$$

Where $A = (a_{i,j})$ and $B = (b_{i,j})$ are $2m \times m$ matrices, $C = (c_{i,j})$ an $m \times 2m$ matrix, and x and y column vectors of size m, each in the ring $\mathbf{R}[u]/(u^r + 1)$. Here $\otimes$ denotes the pointwise product of the two vectors.

Note that with this notation the algorithm can be written as

$$z = \Psi^{-1}(C[(A\Psi(x)) \otimes (B\Psi(y))]).$$

Writing the calculations in this matrix form for Nussbaumer's domain it follows immediately from the definitions that

$$\begin{aligned} z_i &= \sum_{j=0}^{2m-1} c_{i,j}(Ax)_j(By)_j = \sum_{j=0}^{2m-1} c_{i,j}\{\sum_{k=0}^{m-1} a_{j,k}x_k\}\{\sum_{l=0}^{m-1} b_{j,l}y_l\} \\ &= \sum_{k=0}^{m-1}\sum_{l=0}^{m-1} x_k y_l \sum_{j=0}^{2m-1} c_{i,j}a_{j,k}b_{j,l}. \end{aligned}$$

We have already seen that $z_i = d_i + ud_{i+m}$ for $i \in \{0, 1, \ldots, m-1\}$, where $(d_i)_{i=0}^{2m-1}$ is the acyclic convolution[3] of $(x_i)_{i=0}^{m-1}$ and $(y_i)_{i=0}^{m-1}$. As per definition

$$d_i = \sum_{k+l=i} x_k y_l.$$

[3]We calculated this as the cyclic convolution, but the results are identical due to the upper half of $(x_i)_{i=0}^{2m-1}$ and $(y_i)_{i=0}^{2m-1}$ being 0.

Such that

$$z_i = \sum_{k=0}^{m-1} \sum_{l=0}^{m-1} x_k y_l S_{i,k,l}$$

where for $i, k, l < m$

$$S_{i,k,l} = \begin{cases} 1 & \text{if } k + l = i, \\ u & \text{if } k + l = i + m, \\ 0 & \text{otherwise.} \end{cases}$$

Comparing the two forms of z_i gives the equality

$$S_{i,k,l} = \sum_{j=0}^{2m-1} c_{i,j} a_{j,k} b_{j,l}.$$

Now we define two matrices $C' = (c'_{i,j})$ and $A' = (a'_{i,j})$ as

$$c'_{i,j} = \begin{cases} a_{j,0} & \text{if } i = 0, \\ -u^{r-1} a_{j,m-i} & \text{otherwise,} \end{cases}$$

and

$$a'_{i,j} = \begin{cases} c_{0,i} & \text{if } j = 0, \\ u c_{m-j,i} & \text{otherwise.} \end{cases}$$

If we now consider the equation $z' = C'((A'x) \otimes (By))$ we find similar to above that

$$z'_i = \sum_{k=0}^{m-1} \sum_{l=0}^{m-1} x_k y_l S'_{i,k,l}$$

where

$$S'_i = \sum_{j=0}^{2m-1} c'_{i,j} a'_{j,k} b_{j,l}.$$

By Theorem 4 we find that $S_{i,k,l} = S'_{i,k,l}$, so $z_i = z'_i$. Hence we find that for all m-vectors x, y:

$$C((Ax) \otimes (By)) = C'((A'x) \otimes (By)).$$

Consequently, we can generate the same result using these alternative matrices. This could allow for a movement of certain operations from the inverse transform to one of the forward transforms (and vice versa), which could be advantageous if the result of this transform, $A'x$, can be used more than once. In fact, Nussbaumer does not consider A or A' in the operation count, as he assumes that x is fixed and the associated transform can be pre-calculated.

4.5.2 Matrix Form

In order to perform this exchange we need to write the calculations of the algorithm in Nussbaumer's domain in the form $z = C((Ax) \otimes (By))$ for matrices A, B and C. That is, for the input we set the coefficients of the vectors $x_i(u)$ and $y_i(u)$ respectively to $X_i(u_1)$ and $Y_i(u_1)$ of Section 3.1, and as a result we expect $z_i(u)$ to be set to the value of $Z_i(u_1)$ of the same section.

We can achieve this form by setting $A = B$ to the matrix performing the Fourier transform for the input vector appended with m zeroes, and C to the matrices performing the inverse transform described above.

More specifically, we find that the forward Fourier transform, where as in accordance with the optimization of Section 4.3 the first output is not required, can be represented as

$$A = B = \begin{pmatrix} 0 & 0 & 0 & \cdots & 0 \\ \omega^0 & \omega^1 & \omega^2 & \cdots & \omega^{m-1} \\ \omega^0 & \omega^2 & \omega^4 & \cdots & \omega^{2(m-1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \omega^0 & \omega^{2m-3} & \omega^{(2m-3)2} & \cdots & \omega^{(2m-3)(m-1)} \\ \omega^0 & \omega^{2m-2} & \omega^{(2m-2)2} & \cdots & \omega^{(2m-2)(m-1)} \\ \omega^0 & \omega^{2m-1} & \omega^{(2m-1)2} & \cdots & \omega^{(2m-1)(m-1)} \end{pmatrix} \in (\mathbf{R}[u]/(u^r+1))^{2m \times m}.$$

The matrix C calculates the inverse Fourier transform, followed by the correction required that allows for the calculation of the first coefficient to be omitted as explained in Section 4.3 (if we wish to do so), which calculates $(D_i)_{i=0}^{2m-1}$ of Section 3.1. Finally, we need to calculate the $(z_i)_{i=0}^{m-1}$ from this as explained in that section, setting $z_i(u) = D_i + u_1 D_{i+m}$.

If we remove the constant multiplications from the inverse transform and perform it only at the very end, as in the optimization described in Section 4.4, we can write this as the scaled product of three matrices $C = kC_3C_2C_1$, where[4] $k \in \mathbf{R}$ and

$$C_1 = \begin{pmatrix} 0 & \omega^0 & \omega^0 & \cdots & \omega^0 \\ 0 & \omega^{-1} & \omega^{-2} & \cdots & \omega^{-(2m-1)} \\ 0 & \omega^{-2} & \omega^{-4} & \cdots & \omega^{-(2m-1)2} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & \omega^{-(2m-1)} & \omega^{-2(2m-1)} & \cdots & \omega^{-(2m-1)(2m-1)} \end{pmatrix} \in (\mathbf{R}[u]/(u^r+1))^{2m \times 2m},$$

[4] We do not formally define these matrices, as they should be easily understood when compared with the prior descriptions.

$$C_2 = \begin{pmatrix} 1 & 0 & 0 & \cdots & 0 & -1 \\ 0 & 1 & 0 & \cdots & 0 & -1 \\ 0 & 0 & 1 & \cdots & 0 & -1 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & \cdots & 1 & -1 \\ 0 & 0 & 0 & \cdots & 0 & 0 \end{pmatrix} \in (\mathbf{R}[u]/(u^r+1))^{2m\times 2m}, \text{ and}$$

$$C_3 = \begin{pmatrix} 1 & 0 & 0 & \cdots & 0 & u & 0 & 0 & \cdots & 0 \\ 0 & 1 & 0 & \cdots & 0 & 0 & u & 0 & \cdots & 0 \\ 0 & 0 & 1 & \cdots & 0 & 0 & 0 & u & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & 1 & 0 & 0 & 0 & \cdots & u \end{pmatrix} \in (\mathbf{R}[u]/(u^r+1))^{m\times 2m}.$$

Here k represents the accumulated correction factor, C_1 the inverse Fourier transform ignoring the first entry (as per Section 4.3) and without incorporating the scaling factor which has been moved to k. Furthermore, C_2 represents the correction required that allowed the omission of a single length r negacyclic convolution call (also as per Section 4.3), and C_3 calculates z_i given the D_i.

By multiplying these matrices it is seen[5], then, that $C = (c_{i,j}) \in (\mathbf{R}[u]/(u^r+1))^{m\times 2m}$ is defined as follows:

$$c_{i,j} = \begin{cases} 0 & \text{if } j = 0, \\ k[(1+(-1)^j u)\omega^{-ij} - (1+u)\omega^j] & \text{otherwise.} \end{cases}$$

After performing the matrix exchange algorithm as described above, it is easily seen that we have arrived at the algorithm as described in appendix A of Nussbaumer's paper [9], with some additional optimizations with regards to the accumulation of the correction factor.

4.5.3 Implementation

In above form it is clear that the calculations of matrix C' can be performed by a slightly modified fast Fourier transform algorithm followed by a rotation and a sign inversion of coefficients of all but the first polynomial (being the multiplication by the factor $-u^{r-1}$ in the definition of the coefficients of C' above). That is

$$C' = A_t A^T,$$

[5]The result for $i = m-1$ when calculating this is slightly different, but turns out to fall in the 'otherwise' case as well by the identity $\omega^{-(m-1)j} = (-1)^j\omega^j$.

where, informally,

$$A_t = \begin{pmatrix} 1 & 0 & 0 & \cdots & 0 & 0 \\ 0 & 0 & 0 & \cdots & 0 & -u^{r-1} \\ 0 & 0 & 0 & \cdots & -u^{r-1} & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & -u^{r-1} & \cdots & 0 & 0 \\ 0 & -u^{r-1} & 0 & \cdots & 0 & 0 \end{pmatrix}.$$

The A_t operation is trivially performed, whereas A^T can again be calculated through a fast Fourier transform where the first entry is ignored; as such, a polynomial addition can be omitted similar to the previous forward transform.

The work required to apply the matrix A' is not immediately clear; Nussbaumer does not consider this aspect relevant as these operations are not counted toward the number of required operations. As we do count these we still require an efficient method to perform these calculations, which should clearly incorporate a fast Fourier transform in order not to significantly slow down the algorithm.

To perform these calculations efficiently, note that the definition of A' incorporates the multiplication of all but the first elements in the input vector with u, inverting the order of these elements, and finally performing the transpose of C, C^T. That is,

$$A' = C^T C_t.$$

with

$$C_t = \begin{pmatrix} 1 & 0 & 0 & \cdots & 0 & 0 \\ 0 & 0 & 0 & \cdots & 0 & u \\ 0 & 0 & 0 & \cdots & u & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & u & \cdots & 0 & 0 \\ 0 & u & 0 & \cdots & 0 & 0 \end{pmatrix}.$$

Through the identity $[kAB]^T = B^T A^T k$ we find that $C^T = C_1^T C_2^T C_3^T k$. It is easy to see that C_1^T can be calculated by a fast Fourier transform, which can be slightly optimized because the first element is 0, saving one of the polynomial additions in the last iteration. The other matrices are sparse and therefore efficient to perform trivially.

A small further optimizations can be made. Namely, note that the last element of the input vector is not used in C_2^T, hence this value need not be calculated. Note that the multiplication by k should be performed as

	Adds	Mults	CMults
Slow Forward Transform	14 334	0	1 024
Fast Forward Transform	10 208	0	0
Pointwise Multiplication	137 340	11 907	0
Inverse Transform	11 263	0	0

Table 4.2: Number of ring operations (additions, non-constant and constant multiplications) required to calculate the negacyclic convolution after implementing the matrix exchange algorithm. The 'slow' forward transform corresponds to the matrix that was exchanged with the inverse transform matrix.

first step rather than last, since the input vector has only m elements while the output has $2m$; performing it at a later time doubles the amount of multiplications required.

4.5.4 Results

The results of implementing the matrix exchange algorithm into the previous version, as implemented in the test 'Nussbaumer version' in the associated source code, are shown in Table 4.2.

It is noteworthy that not only the total number of additions in the forward and inverse transforms are greater than in the previous version; the total number of additions in the recursion step is, as well. This comes from the fact that the same exchange has been implemented for the recursive calls of the algorithm. It would be possible to use the same recursive calls as in the previous version, in which the algorithm only requires 125 extra additions. On the other hand, for each time the transformed polynomial is re-used, 14 334 - 10 208 = 4 126 additions are saved when compared to re-using the original transformed input.

This table allows for easy comparison with Nussbaumer's paper. The slow forward transform is considered to be free in Nussbaumer's paper, so ignoring this row the number of multiplications matches exactly that of the predictions by Nussbaumer [9]. More additions than predicted are required, however: while it is easy to verify that the forward fast Fourier transform matches the number of predicted operations (Appendix A, Step 3 of his paper), the inverse transform requires 54 more additions. This can be explained by the fact that Nussbaumer does not count sign inversions as additions, while we do. Furthermore, the pointwise multiplication is more expensive due to this effect propagating into the recursive calls.

As indicated in Nussbaumer's paper it is also possible to perform more preparative steps from the pointwise multiplication at the forward transform

	Adds	Mults	CMults
Slow Forward Transform	85 398	0	1 024
Fast Forward Transform	34 967	0	0
Base Case	7 938	11 907	0
Inverse Transform	44 842	0	0
Total	173 145	11 907	1 024

Table 4.3: Number of ring operations (additions, non-constant and constant multiplications) required to calculate the negacyclic convolution after implementing the matrix exchange algorithm, where the preparative steps for the pointwise multiplications have been moved to the forward transforms.

stage. So far we considered the algorithm to run recursively: the pointwise multiplication required again transforms that were dependent on only one of the inputs. As such, these additions can also be moved into the forward transform steps.

The results, implemented in the same test as the previous, are shown in Table 4.3. Note that the total number of operations does not change, but they are merely moved to different stages. Rather than the 4 126 additions that can be saved by re-using the transformed input as explained previously, the saved number of additions per re-use becomes 85 398 - 10 208 = 75 190, which is nearly 45% of the total additions required for a full run of the algorithm.

4.6 Missing Optimization

We omit a single optimization Nussbaumer suggests. He states that "the number of additions can still be slightly reduced by replacing two of the interpolation polynomials $u_2 - u_1^k$ by the simpler polynomials u_2, $1/u_2$." [9][6]

The means to this optimization, unfortunately, remains unclear. In fact, in the remainder of the article he no longer refers to this: in appendix A he counts the ring operations without taking this optimization into account. As such, this optimization falls outside of the scope of this thesis.

4.7 Comparison

The number of operations required for Nussbaumer's algorithm as implemented by BCNS [7] are shown in Table 4.4. As the stages of the algorithm

[6]Notation adapted to ours.

Additions	105 520
Non-constant Multiplications	65 536
Constant Multiplications	12 288

Table 4.4: Number of ring operations (additions, non-constant and constant multiplications) required for the Nussbaumer algorithm as implemented by BCNS.

	Adds	Mults	CMults
Forward Transform	15 360	0	6 144
Pointwise Multiplication	0	1 024	0
Inverse Transform	15 360	0	6 144
Total	46 080	1 024	18 432

Table 4.5: Number of ring operations (additions, non-constant and constant multiplications) required for the NTT algorithm as implemented in New Hope. The row 'total' provides the operation count for the full calculation of a negacyclic convolution; note that two forward transforms are required.

are not implemented separately we list only the total number of ring operations for a full calculation of the negacyclic convolution.

This implementation does not use Nussbaumer's algorithm for any but the first level of recursion: for the second level, the schoolbook approach is used for calculating length 32 negacyclic convolutions. For fairness of comparison we will consider an alternative implementation using the same; also, since the parts are combined into one method the matrix exchange optimization is not possible, so we will assume this has not been implemented as this would come at the expense of extra additions without any gains in this context.

As we will see in Section 5.6.1 the calculation of length 32 negacyclic convolutions using the schoolbook method requires 992 additions and 1 024 multiplications, and $2m - 1 = 63$ such convolutions are to be calculated. In our implementation this results in $2 \cdot 10\,208 + 15\,264 + 63 \cdot 992 = 98\,176$ additions, $63 \cdot 1\,024 = 64\,512$ non-constant multiplications and 1 024 constant multiplications. As can be seen when compared with Table 4.4 we were more efficient for each type of operation, but most significantly so for constant multiplications, which we managed to reduce by 92%.

We now consider the NTT as implemented in New Hope; the number of ring operations, as implemented in the test 'NewHope version' in the associated source code, are shown in Table 4.5.

For n multiplications with the same polynomial using NTT one forward transform can be used, hence the total number of operations required are $15\,360 + n \cdot 30\,720$ additions and $6\,144 + n \cdot 14\,336$ multiplications. On the other hand, the implementation of Nussbaumer's algorithm requires $85\,398 + n \cdot 87\,747$ additions and $1\,204 + n \cdot 11\,907$ multiplications.

It is obvious that Nussbaumer's algorithm requires somewhat fewer multiplications, but significantly more additions. While the ratio of cost of a multiplication to an addition differs per architecture and implementation it is unlikely this ratio is ever high enough on modern systems to make Nussbaumer's algorithm more efficient than the NTT in terms of time required to perform the additions and multiplications.

Furthermore, most multiplications for NTT are constant multiplications, while Nussbaumer's method uses mostly non-constant multiplications. This is yet another advantage for NTT, since constant multiplications may be implemented more efficiently than non-constant multiplications. For example, in the case of New Hope integers are stored in Montgomery domain [23] (see [8, Section 7.2]), and the translation to this domain can be pre-calculated in the case of constant multiplications.

4.8 Conclusion

The implementations publicly available lack all or most of the optimizations described here, even though merely one of these implementations is vectorized and as such the others could easily implement these. We have shown these optimizations are able to drastically reduce the number of operations required when compared with the implementation by BCNS [7].

Nevertheless, we have shown that NTT still greatly outperforms Nussbaumer's algorithm in terms of ring operations. We may gather from this that Nussbaumer's algorithm would need to be deprecated. However, as we will see in the next chapter, it turns out that the use of Nussbaumer's algorithm in the context of New Hope provides an additional freedom that allows for significant optimizations that are not possible with NTT.

The final recursive implementation of the algorithm, slightly modified for aesthetic reasons, can be seen in Appendix B. For brevity we omit the iterative part corresponding to the moving of preparative steps in the recursion calls to the forward transform.

Chapter 5

Vectorization

5.1 Introduction

Whereas the previous chapter introduced the ring operation count as a measure of work required for Nussbaumer's algorithm, the results are not directly transferable to implementations on most modern hardware. The reason for this is that modern hardware often supports vectorization of certain instructions, allowing multiple independent operations of the same kind to be done in parallel. As we will see in this chapter, exploiting such support the number of clock cycles required can be reduced drastically.

In this section we present an AVX2 implementation, written in Assembly language, of Nussbaumer's algorithm. For sake of completeness we will start by giving a brief overview of AVX2. This is followed by details about the implementation and finally its performance with respect to NTT. For our implementation we shall focus on the Intel Haswell processor architecture.

By no means do we attempt to give an optimal implementation here; the ideal answers to the questions posed in this chapter themselves would require thorough studies and, as such, further improvements will be possible. However, we do try to supply a well-considered and optimized implementation that can be used for an initial comparison between AVX2 implementations of NTT (as given in New Hope) and Nussbaumer's algorithm.

5.2 AVX2

AVX2 [24], which stands for Advanced Vector Extensions 2, is an extension of the x86 instruction set supported by several CPUs from both Intel and AMD, the first of which, the Intel Haswell, released in 2013. AVX2, itself an extension of AVX, is an instantiation of the concept of Single Instruction, Multiple Data (SIMD): that is, it allows a single instruction to act on multiple fields of data simultaneously.

In the case of AVX2 sixteen 256-bit registers, named `ymm0-15`, were

introduced, whose content can consists of vectors of either floating point numbers or integers of various sizes. Each of these registers can then be acted on by a variety of instructions, where the nature of these vectors are often encoded in the instruction: examples are `vpaddw` and `vaddpd`, which perform a pointwise addition for two pairs of vectors consisting of word-sized integers and double precision floating point numbers respectively.

Many of these added instructions perform pointwise operations of multiple vectors, such as the mentioned addition instruction, but other operations are also included. Notable examples are "horizontal" instructions, which can operate on all of the entries of a single vector (for example finding its maximum), and instructions to rearrange or combine vectors.

Through AVX2 it is possible to parallelize the calculation of a single operation on multiple fields of data - as many as 32 for byte-sized integers - while the instructions achieve similar speed in terms of clock cycles when compared with the execution of their non-vectorized original of the x86 instruction set, as can be seen in Agner Fog's instruction tables [20]. Exploiting this, an algorithm may be able to significantly decrease the execution time of certain algorithms.

5.3 Parameter Choice

In order to fairly evaluate the usefulness of Nussbaumer's algorithm we shall compare its performance for an optimized implementation comparable to that of New Hope's NTT algorithm. As such we use the same $N = 1024$ for some chosen q with $\mathbf{R} = \mathbb{Z}/q\mathbb{Z}$, where N is the length of the sequences to calculate the negacyclic convolution for and $\mathbf{R}$ the ring of the coefficients.

Note that the NTT algorithm requires $q \equiv 1 \pmod{2N}$ "so that the number-theoretic transform (NTT) can be realized efficiently" [8]. In fact, as "the security level grows with the noise-to-modulus ratio, it makes sense to choose the modulus as small as possible, improving compactness and efficiency together with security," [8] q is chosen to be minimal under said condition.

Nussbaumer's algorithm, on the other hand, does not suffer this limitation. As a consequence we can choose an even smaller q, further improving the security[1] of New Hope. In fact, q does not even need to be prime; it merely needs to be odd[2]. BCNS, for one, decided to use a composite q; in accordance with their paper we will not restrict it to a prime number. The discussion of the security implications of this choice of q is out of the scope of this thesis.

[1] The compactness of the packets used in key exchanges could also be improved, but this is considered irrelevant for our comparison.

[2] As seen above, the only requirement for $\mathbf{R} = \mathbb{Z}/q\mathbb{Z}$ for the algorithm is that $2m$ has an inverse, which is true if and only if q is odd.

In order to allow for reduction to be performed efficiently, we choose $q = 2^k - 1$ for a certain k, such that no division is required for reduction. In fact, reduction can then be performed by shift, bitwise-AND and addition operations as we will show later.

We pick $k = 11$, that is, $q = 2047$, in order to keep q relatively close to the original to prevent alternative issues, such as increasing the error rate[3]. Should one not wish to depend on the assumption in BCNS that a composite q is acceptable, one could build a similar implementation with $k = 13$, for which q is a Mersenne prime in the same form, still lower than the value used in New Hope. This would somewhat slow down the implementation when compared to the version we present here however, as reductions are required more frequently.

5.4 Integer Representation

Since the numbers (easily) fit in 16 bits, but not in 8, we treat each AVX2 register as a 16-tuple of 16-bit integers to maximize the number of simultaneous operations performed by each instruction. The extra bits allow us to amortize the reduction over multiple addition instructions. To exploit this we keep track of the possible range of all integers through static analysis, and perform the reduction step for intermediate results only when an overflow could be caused at the next.

Furthermore, as the algorithm requires a large number of subtractions we treat the integers as signed in most cases; only when it is relevant for reduction we add multiples of q in order to get an equivalent but positive number that is then treated unsigned.

In the following we discuss the strategy of reduction of said integers.

5.4.1 Reduction After Addition

In order to reduce the vectorized integers before an overflow may happen, we add a multiple of q such that the result is still correct modulo q and the result is always a positive integer. After this step the sign bit is no longer required but an overflow into this bit can occur, which both forces and allows us to treat the integers as unsigned.

Following this each entry in the resulting vector register is a 16-bit integer x, say $x = x_l + 2^{11}x_h$, where $x_l, x_h \in \mathbb{N}$ with $x_l < 2^{11}$ and $x_h < 2^5$. This is then reduced to $x_l + x_h \equiv x_l + 2^{11}x_h \equiv x \pmod{q}$. These calculations can trivially be performed by a shift (to find x_h), bitwise-AND (to find x_l) and an addition operation (to find $x_l + x_h$).

[3]Note that one of the contributions of New Hope improves the analysis of the failure probability which allowed q to be decreased in the first place.

One such reduction step does not provide an output that is at most 11 bits: note that $x' = x_l + x_h < 2^{11} + 2^5 < 2^{12}$, so the output is guaranteed to contain at most 12 bits. For intermediate results the value is not reduced further and is treated as usual (in particular, note that the sign bit is guaranteed to be 0, so we can again interpret this integer as signed).

At the end of the algorithm the numbers need to be fully reduced: failure to do so could potentially disclose information about the input of polynomials which in the context of New Hope could pose a security risk, and reducing it allows for a tighter packing for transmission. To achieve this, we perform another reduction: say $x' = x'_l + 2^{11}x'_h$ is the output of a partial reduction, where $x'_l \in \mathbb{N}$ with $x'_l < 2^{11}$ and $x'_h \in \{0, 1\}$, such that the reduction step produces $x'' = x'_l + x'_h \equiv x \pmod{q}$. Since $x' < 2^{11} + 2^5$ if $x'_h = 1$ then $x'_l + 2^{11} < 2^{11} + 2^5$, and if $x'_h = 0$ then $x'_l + x'_h < 2^{11}$; in either case $x'' = x'_l + x'_h < 2^{11}$.

Finally, to perform a full reduction, we require that $x'' < q$; that is, if $x'' = q$ it needs to be reduced further to $x'' = 0$. For this we use the `vpcmpeqw` instruction to compare equality with q. As the individual entries in the destination `ymm` register of this comparison will be filled with either ones or zeroes on equality and inequality respectively, subtracting these from the result combined with a bitwise-AND to select the proper bits produces the wanted form.

5.4.2 Reduction After Multiplication

The results of the required non-constant multiplications generally do not fit in 16-bit integers; even the product of two 11-bit integers could take up 22 bits. This problem is mirrored in the AVX2 instructions by having a separate instruction to store the lower and upper halves of a multiplication or alternatively, in the case of the `vpmaddwd` instruction, to store only the sum of adjacent multiplications.

While the `vpmaddwd` instruction performs not only sixteen full multiplications but also eight additions, and as such will likely result in a quicker implementation when properly incorporated, we opted for a more intuitive implementation by retrieving the lower and upper half of the multiplication separately and combining the two while reducing the results.

Incorporating the `vpmaddwd` instruction is advisable as future work, though it may be seen implemented in its SSE counterpart for the 'Sieving for Shortest Vectors in Ideal Lattices' paper [17, 18].

Here, like in the reduction after addition, the reduction is amortized over several additions of the upper halves of the multiplication which are guaranteed not to overflow, as the addition is cheaper than the reduction. More specifically, the inputs for any multiplication are guaranteed to contain at most 15 bits[4] and be non-negative. Let x_1, x_2 be such inputs. Then

[4]This number of bits turned out to be the natural size of the outputs of the algorithms feeding these integers, when maximally postponing reduction steps.

$x_1 x_2 \leq 2^{30}$, so for $y = x_1 x_2 = y_l + 2^{16} y_h$, where y_l and y_h represent the low and high half 16-bit integers of the result respectively, we see that $y_h < 2^{14}$. As the sum of reductions is equal to the reduction of a sum, both modulo q, we can add multiple of such higher halves; due to the upper bound these are guaranteed not to overflow as unsigned integers when four are added up. Incorporating these into the reduced result can thus be postponed. The resulting upper half is then reduced by trivial appropriate shift, AND and addition instructions.

For the lower halves this postponing is not possible: the result requires a full 16 bits, and as such may cause an overflow after any such addition.

Note that the results of these non-constant multiplications are always intermediate and as such not fully reduced. The constant multiplications, by the resulting constant factor $\frac{1}{2m}$, turns out to be far simpler: as $\frac{1}{2m} = 32$, this is implemented as a rotation of bits, and reduction according to the addition strategy is used instead.

5.5 Operation Cost

In the remainder of this chapter we will discuss several decisions made on the trade-off between addition and multiplication operations. To do so, we need to discuss the relative cost of these operations. For this we interpret the instruction tables from Agner [20]; specifically those concerning the Haswell processor architecture for which we optimize our implementation.

Whereas several measures of the speed of these instructions exist, we focus on the optimistic side, namely the reciprocal throughput: "the average number of core clock cycles per instruction for a series of independent instructions of the same kind in the same thread." [20] In practice subsequent instructions may not be independent causing these measurements to be lower than in practice, but they do give a general indication of the clock cycles required for an operation.

We see that the applicable `vpaddw` instruction has a reciprocal throughput of 0.5 clock cycles, while each of the two halves of the multiplication instruction has a reciprocal throughput of 1 clock cycle. This implies a 1 to 4 ratio in number of clock cycles for additions to multiplications.

However, this view is incomplete as it does not measure reductions. Where reductions after addition steps can be amortized as seen above, measures need to be taken to handle the output of the two multiplication operations: in any case, an operation is required to combine the two results into a single register. Hence in practice a multiplication step will require more than 4 times the amount of clock cycles an addition operator costs.

The exact balance is outside the scope of this thesis as the number is only used as an estimate. As such, we assume a multiplication operation to require approximately the same number of clock cycles as 5 addition operations.

	Adds	Mults	CMults
Nussbaumer's algorithm	2 180	189	32
Schoolbook method	992	1 024	0
Karatsuba's algorithm	1 145	243	0

Table 5.1: Number of ring operations (additions, non-constant and constant multiplications) required to calculate the negacyclic convolution of a sequence of length 32.

5.6 Secondary Method

Recall that in the previous chapter we recursed into Nussbaumer's algorithm, as suggested by Nussbaumer. In the following we argue the secondary method we use, and the structure the input takes in the vector registers.

5.6.1 Secondary Algorithm

The pointwise multiplications required in Nussbaumer's algorithm with $N = 1024$ are themselves negacyclic convolutions of sequences with 32 coefficients. So far we used the algorithm recursively in accordance with Knuth and Nussbaumer, even though Nussbaumer's algorithm may be less efficient for small instances of the problem. To explore some alternatives, the 'secondary method' test in the associated source code lists the number of additions and multiplications required for recursion into the schoolbook method and Karatsuba's algorithm.

The results are shown in Table 5.1. Unlike in Nussbaumer's original paper, forward transformations are counted towards the operation count, as their results can not generally be pre-calculated in the instance of New Hope; in fact, as shown later, they can be used twice at best. As such, even though some of the operations can be combined into a preparative stage, the gains from a matrix exchange algorithm are limited, so we will ignore the possibility in this section.

The schoolbook method requires 1 188 fewer additions, but 835 more non-constant multiplications than recursing into Nussbaumer's method, and thus introduces a significant increase in estimated clock cycles. Karatsuba's method on the other hand only costs 54 additional non-constant multiplications, so, assuming the 5-to-1 ratio of clock cycles described above, 270 additions could be performed in an approximately equal time. However, Karatsuba saves significantly more than that in number of additions, as it requires 1 035 fewer than Nussbaumer's method. In our implementation we omit the small optimization discussed for negacyclic convolutions using Karatsuba's method in Section 2.2, so the number of additions can be even

be slightly reduced further.

According to these metrics (and even for significantly larger multiplication to addition clock cycle ratios), Karatsuba's method would likely outperform Nussbaumer's method, and as such is as per our estimation the preferred method to use to calculate the length 32 negacyclic convolutions.

This is contrary to the results given in 'Practical Fast Polynomial Multiplication' [25], which states that "it may be seen that at degree 31 Karatsuba's algorithm takes three times longer than the classical method." However, it continues to state: "on reason for the poor performance of the algorithm was that it had to labour very hard to perform many operations both in the form of procedure calls and arithmetic operations close to the base of its recursion. Therefore, one way to improve the algorithm would be to stop the recursion at some small degree and perform this multiplication classically."

While this article would suggest that for the 32 coefficient polynomials in this case the schoolbook method would outperform Karatsuba's method, it is relevant that said article stems from 1976, well before the introduction of AVX2 or predecessors, or many other improvements such as increased cache-size that can affect these results. As such, these results should not be directly adopted in our setting.

That does not mean that the article has no merit; Karatsuba could indeed be outperformed by the schoolbook method for small-degree polynomials. For this reason we fall back to the schoolbook method for polynomials with 4 coefficients. As stated before, this need not be the fastest choice, but deciding the optimal methods falls outside the scope of this thesis.

5.6.2 Parallelization Direction

The parallelization of the polynomial manipulation instructions through AVX2 can be done either horizontally or vertically. The parallelization is called horizontal if a single register contains subsequent coefficients of the same polynomial, while it is called vertical if a single register contains the same coefficient for subsequent polynomials.

Each of these directions will cause distinct advantages and disadvantages during implementation of Karatsuba's algorithm. The vertical parallelized implementation itself has very little overhead, as the multiplication can be done similar as one would do without any parallelization and the additional polynomials that are represented in the registers - in our case fifteen - would be multiplied for free. The horizontal version on the other hand does require overhead, as there are no instructions in AVX2 to easily perform the required operations on a single register. For example in Karatsuba's method, adding part of a register with another part of itself takes additional operations to permute the coefficients in the inputs.

Note that vertical parallelization of Nussbaumer's algorithm is infeasible:

at different stages of the fast Fourier transform each polynomial is rotated a different amount, so collecting the right coefficients to add is expensive. Here, horizontal parallelization is obligatory in order to achieve efficiency. This also means that the natural order to store the output of Nussbaumer's forward transform algorithm or, similarly, the input for the inverse transform algorithm is horizontal.

This is where one of the disadvantages of vertical parallelization comes from: Nussbaumer's transform algorithms require the inputs and outputs from Karatsuba's method to be changed in direction. In other words the input and output needs to be transposed, which does impose an additional overhead. Second, as the vertical parallelization in our case multiplies a multiple of sixteen polynomials, the optimization in Section 4.3, allowing for one convolution to be removed, is no longer applicable as the removed convolution is calculated for free.

From this it is not apparent which version will be faster. We have decided to go with vertical parallelization, as this further simplifies Nussbaumer's algorithm by making said optimization superfluous.

5.6.3 Optimizations

In Chapter 4 we considered optimizations proposed by Nussbaumer. We must further consider whether these optimizations have merit in our vectorized implementations: we have already seen that, given our choices, the removal of a convolution of Section 4.3 is not applicable.

The multiplication factor accumulation of Section 4.1 is trivially applicable, but the recursion factor accumulation of Section 4.4 only partially: while it is still better to postpone the multiplication as described in that section, there is no longer a factor in the used recursion method as Karatsuba's algorithm is used instead of Nussbaumer's algorithm. As such, no factor multiplication from the algorithm calculating the length r negacyclic convolution can be postponed.

Using the horizontal vectorization of Nussbaumer it is not possible to combine the rotation of polynomials with addition to another polynomial, as it is when arrays of scalars are used. Instead the rotation has to happen explicitly. As such, the modified inverse transform of Section 4.2 is irrelevant, and there is no longer any preference between the two versions. We still implement the DIT FFT for consistency with the forward transform used.

The matrix exchange algorithm of Section 4.5 remains to be discussed. As we will see later most transformed polynomials are used only once and, without imposing additional constraints on the New Hope algorithm, the others are used at most twice. That is, the transformed input can be reused only once at maximum.

Note that no matrix exchange is required to perform the preparative

steps of the pointwise multiplication (that do not depend on the other polynomial input) at the forward transform. In the recursive strategy used by Nussbaumer this amplifies the effect of the matrix exchange algorithm, as each level of recursion moves work to the preparation. As we use Karatsuba as secondary method, and no matrix exchange can be performed for this algorithm, the only gain is that of the instance of Nussbaumer's algorithm with $N = 1024$.

We have seen that this optimization was able to save 4 126 additions per re-use, with an added cost of 125 additions. However, these results were considering the removal of convolution of Section 4.3, which we omit in our implementation. This makes the inverse matrix significantly cheaper, further reducing the gains of this optimization.

Furthermore, the constant multiplications at the inverse transform do not require a matrix exchange to move to the forward transform: they can trivially be moved to the initial step, as can be seen from the definition of the negacyclic convolution, since calculating the negacyclic convolution with a fixed polynomial preserves scalar multiplication.

It would be possible to somewhat speed up the algorithm exploiting these facts at the cost of complicating the code. However, combining the facts the gains are minimal and they can not commonly be exploited as most transforms can either not at all or merely once be re-used we chose to omit this optimization.

In our implementation we do perform the preparative steps for the Karatsuba algorithm, which do not depend on the secondary input, at the forward transform. This omits these steps from being repeated in the case of re-use, without significantly complicating the code.

5.7 Application to New Hope

As can be seen in protocol 3 of 'Post-Quantum Key Exchange – a New Hope' [8], New Hope's key-exchange algorithm requires 4 forward transformations, 2 inverse transformations and 4 pointwise multiplications.

Said algorithm will not work directly in the case of Nussbaumer for two reasons. First of all, note that the transformations in New Hope are one-to-one, as the forward and inverse transformations are easily seen to be inverses of each other. As the polynomial $\mathbf{a}$ is uniform noise and, due to the bijective property, "the NTT transforms uniform noise to uniform noise" [8], the two forward transforms on these inputs can be skipped as long as both parties agree.

The transformations in Nussbaumer's algorithm are not bijections, which is easy to see as the cardinality of the codomain is double that of the domain. As such, on both ends an additional forward transform must be performed.

Second, the algorithm in New Hope communicates the polynomials in

the NTT domain. While this is still possible using Nussbaumer's algorithm it would, due to the double size of the cardinality of Nussbaumer's domain, double the amount of bits required to communicate the polynomial. As we wish the implementation of Nussbaumer's algorithm to impose no negative side-effects on the algorithm, the polynomials should be sent in their original domain in order to not increase the bandwidth requirement. In fact, due to the fewer number of bits required to represent the coefficients the polynomial could even be packed tighter, reducing the bandwidth.

Protocol 5.1 shows the corresponding protocol that is no more demanding on network resources than the original. Here six forward transformations, four inverse transformations and four pointwise multiplications are required: this is two more forward transformations and two more inverse transformations than the original.

As we can see most of the polynomial transforms ($\hat{\mathbf{s}}$ and $\hat{\mathbf{u}}$ on the server side and $\hat{\mathbf{a}}$ and $\hat{\mathbf{b}}$ on the client side) are only used once, whereas the others, $\hat{\mathbf{a}}$ on the server side and $\hat{\mathbf{t}}$ on the client side are used exactly twice.

5.8 Theoretical Lower Bound

In order to count the number of instructions, the GDB script shown in Listing 5.1 is used. This can be used by executing GDB as

```
gdb -ex 'set $startaddr=StartAddress' -ex 'set $endaddr=
    EndAddress' -x ScriptFile BinaryFile
```

where *StartAddress* and *EndAddress* represent the program counter for the first and last instruction of the code to count operations for, *ScriptFile* the file containing the GDB script and *BinaryFile* the binary file to get the instruction count for. GDB will then output all instructions executed with the program counter starting at *StartAddress* until it reaches *EndAddress*.

```
break *$startaddr
set pagination off
display/i $pc
run
while($pc != $endaddr)
  stepi
end
quit
```

Listing 5.1: GDB script responsible for dumping executed instructions.

The number of AVX2 instructions, ignoring loading, storing and moving instructions, are shown in Table 5.2. The corresponding total number of micro-operations per set of ports are shown in Table 5.3. As previously

mentioned the preparative steps for the pointwise multiplications are listed under the forward transform.

We now consider for the three sections of the algorithm the minimum amount of clock cycles required for these instructions as a measure of a lower-bound for the time required to execute our algorithm. To do so, we find the bottleneck CPU port(s), assuming they are ideally distributed, and its corresponding operation count. This indicates the minimum number of clock cycles the port requires for the total of the algorithm. The results are listed in Table 5.4; we omit the proofs, as they are simple to verify.

These lower bounds, then, indicate the minimum running time of each stage of the algorithm in clock cycles, assuming the provided implementation.

5.9 Benchmark and Comparison

As in accordance with the New Hope paper [8], we perform the benchmarks on an Intel Core i7-4770K (Haswell) processor, running at 3491.924 MHz, with Hyperthreading and Turbo Boost disabled. Also, the code was compiled using g++ and gcc version 4.9.2 and GNU assembler version 2.25, where for the C(++) code the flags '-O3 -fomit-frame-pointer -march=corei7-avx -msse2avx' were used, again in accordance with New Hope.

The benchmarks, as shown in Table 5.5, give the mean and average (in parenthesis) running time in clock cycles of each relevant function, over 1,000 runs. Only the AVX2 implementations are considered. The number of clock cycles was determined using the `rdtsc` instruction; the routines from New Hope are used for this purpose.

As seen when comparing with the results of the previous section, we have achieved speeds of 57%, 57% and 70% of the theoretical lower bounds for respectively the forward transform, inverse transform and the pointwise multiplication.

For a single polynomial multiplication the total average time required using the NTT algorithm is 24 872 cycles, while for Nussbaumer's algorithm the running time is a mere 15 660 clock cycles, providing a speed up of about 37% with respect to NTT.

As discussed in the Section 5.7 more transformations are required when Nussbaumer's algorithm is applied to the New Hope key exchange protocol. Given these running times we can estimate the running time of the part of the protocol calculating the negacyclic convolution using NTT to be about $4 \cdot 10\,536 + 2 \cdot 11\,620 + 4 \cdot 2\,716 = 76\,248$ clock cycles, while similar for Nussbaumer's algorithm would be about $6 \cdot 2\,348 + 4 \cdot 2\,724 + 4 \cdot 10\,588 = 67\,336$. In this case, considering the additional transformations required, Nussbaumer's algorithm would still provide a speed up of approximately 12%.

		Instruction Count		
	Ports	Forward	Inverse	Pointwise
`vpaddw`	p15	1 136	1 216	6 808
`vpsubw`	p15	418	578	1 056
`vpmullw`	p0			1 728
`vpmulhuw`	p0			1 728
`vpand`	p015	256	448	3 236
`vpxor`	p015	1	1	
`vpor`	p015		64	
`vpsrlw`	p0 p23	256	512	3 236
`vpsllw`	p0 p23		64	756
`vpcmpeqw`	p15		64	
`vpalignr`	p5	136	200	
`vperm2i128`	p5	388	452	
`vpunpcklwd`	p23 p5	96	96	
`vpunpckhwd`	p23 p5	96	96	
`vpunpckldq`	p23 p5	96	96	
`vpunpckhdq`	p23 p5	96	96	
`vpunpcklqdq`	p5	96	96	
`vpunpckhqdq`	p5	96	96	

Table 5.2: Number of non-moving AVX2 instructions required for Nussbaumer's algorithm. Unused instructions for certain parts of the algorithm are left blank. The 'ports' column indicating the execution ports of the processor used, as seen for the Haswell processor architecture by Agner [20]: each column (separated by space) indicates the usage of a port, and a 'p' followed by multiple digits indicates exactly one of the following ports must be used.

Ports	Forward	Inverse	Pointwise
0	256	576	7 448
5	1 100	1 228	
1 or 5	1 554	1 858	7 864
2 or 3	640	960	3 992
0, 1 or 5	257	513	3 236

Table 5.3: The number of micro-operations to each group of ports for Nussbaumer's algorithm.

Stage	Bottleneck	Lower Bound
Forward	ports 1 and 5	1 327
Inverse	ports 1 and 5	1 543
Pointwise	port 0	7 448

Table 5.4: The lower bounds on the number of micro-operations required, together with the CPU port(s) that form the bottleneck.

It is noteworthy that the pointwise multiplication of New Hope's implementation has not been vectorized. However, even if we assume this step can be optimized to a no-op in the case of NTT, but still considering it for Nussbaumer, we see that the latter is merely 3% slower. Note that of course this assumption is unrealistic: some clock cycles will be required for this step. Without an optimized implementation for this step we can only conclude the results will not deviate much in terms of speed when such an optimized implementation is provided, but are likely somewhat favorable for Nussbaumer's algorithm in terms of speed.

	NTT	Nussbaumer
Forward Transform	10 536 (10 579)	2 348 (2 372)
Inverse Transform	11 620 (11 621)	2 724 (2 727)
Pointwise Multiplication	2 716 (2 719)	10 588 (10 623)

Table 5.5: Median and average (in parenthesis) running times in clock cycles of portions of the NTT and Nussbaumer's algorithm.

Alice (server)		**Bob (client)**
$seed \xleftarrow{\$} \{0,\ldots,255\}^{32}$		
$\mathbf{a} \leftarrow \text{Parse}(\text{SHAKE-128}(seed))$		
$\mathbf{s}, \mathbf{e} \xleftarrow{\$} \psi_{16}^n$		$\mathbf{s}', \mathbf{e}', \mathbf{e}'' \xleftarrow{\$} \psi_{16}^n$
$\hat{\mathbf{a}} \leftarrow \text{Transform}(\mathbf{a})$		
$\hat{\mathbf{s}} \leftarrow \text{Transform}(\mathbf{s})$		
$\mathbf{b} \leftarrow \text{Transform}^{-1}(\hat{\mathbf{a}} \circ \hat{\mathbf{s}}) + \mathbf{e}$		
$m_a \leftarrow \text{encodeA}(seed, \mathbf{b})$	$\xrightarrow{m_a}$	$(\mathbf{b}, seed) \leftarrow \text{decodeA}(m_a)$
		$\mathbf{a} \leftarrow \text{Parse}(\text{SHAKE-128}(seed))$
		$\hat{\mathbf{a}} \leftarrow \text{Transform}(\mathbf{a})$
		$\hat{\mathbf{b}} \leftarrow \text{Transform}(\mathbf{b})$
		$\hat{\mathbf{t}} \leftarrow \text{Transform}(\mathbf{s}')$
		$\mathbf{u} \leftarrow \text{Transform}^{-1}(\hat{\mathbf{a}} \circ \hat{\mathbf{t}}) + \mathbf{e}'$
		$\mathbf{v} \leftarrow \text{Transform}^{-1}(\hat{\mathbf{b}} \circ \hat{\mathbf{t}}) + \mathbf{e}''$
		$\mathbf{r} \xleftarrow{\$} \text{HelpRec}(\mathbf{v})$
$(\mathbf{u}, \mathbf{r}) \leftarrow \text{decodeB}(m_b)$	$\xleftarrow{m_b}$	$m_b \leftarrow \text{encodeB}(\mathbf{u}, \mathbf{r})$
$\hat{\mathbf{u}} \leftarrow \text{Transform}(\mathbf{u})$		
$\mathbf{v}' \leftarrow \text{Transform}^{-1}(\hat{\mathbf{u}} \circ \hat{\mathbf{s}})$		
$\mathbf{v} \leftarrow \text{Rec}(\mathbf{v}', \mathbf{r})$		$\mathbf{v} \leftarrow \text{Rec}(\mathbf{v}, \mathbf{r})$
$\mu \leftarrow \text{SHA3-256}(\mathbf{v})$		$\mu \leftarrow \text{SHA3-256}(\mathbf{v})$

Protocol 5.1: Protocol for New Hope, akin to protocol 3 in the New Hope paper [8], except that it uses Nussbaumer's algorithm for calculating negacyclic convolutions. Conventions of the original have been used, although here Transform and Transform^{-1} indicate the forward and inverse Nussbaumer transformations respectively, elements in Nussbaumer's domain are denoted with a hat and $\circ$ means the pointwise product in the corresponding Nussbaumer domain.

Chapter 6

Conclusion

We have seen that Nussbaumer's negacyclic convolution algorithm is rarely implemented and, in the few instances it has been implemented, it lacks the optimizations specified by Nussbaumer. In particular, while a vectorized implementation exists, its performance is not compared to similar algorithms; also, the specific instance uses a different choice of parameters and is therefor not directly applicable to or comparable with our implementation.

As we have shown, Nussbaumer's algorithm requires significantly more ring operations to calculate the same negacyclic convolution when compared to NTT. We conclude that any implementation using the same parameters for Nussbaumer and NTT would most likely result in a favorable outcome efficiency-wise for the latter.

However, NTT is limited in its use in the way that it restricts the allowable parameters, namely it restricts freedom of choice of ring over which the coefficients are chosen. Nussbaumer's algorithm on the other hand works with far less restriction on this ring; only that $2m$ needs to be invertible.

In the context of New Hope the ring can be chosen freely. We have modified this ring to make reductions more efficient. We have shown that in this context Nussbaumer's algorithm turns out not to merely enable a greater security for New Hope, but even performs better than the commonly used number-theoretic transforms.

This conclusion can be transferred to a generic subset of ring learning with error algorithms: if the choices of parameters cause a negacyclic convolution to be a central part of the algorithm (as in New Hope and BCNS), then preferring Nussbaumer's algorithm over NTT gives additional freedom for the chosen ring. This choice may in general lead to more efficient and secure algorithms.

To conclude, we have seen that implementing Nussbaumer's algorithm in the context of the ring learning with errors key exchange provides a new hope for Nussbaumer's algorithm.

Bibliography

[1] T. Lanting, A. J. Przybysz, A. Y. Smirnov, F. M. Spedalieri, M. H. Amin, A. J. Berkley, R. Harris, F. Altomare, S. Boixo, P. Bunyk, N. Dickson, C. Enderud, J. P. Hilton, E. Hoskinson, M. W. Johnson, E. Ladizinsky, N. Ladizinsky, R. Neufeld, T. Oh, I. Perminov, C. Rich, M. C. Thom, E. Tolkacheva, S. Uchaikin, A. B. Wilson, and G. Rose, "Entanglement in a quantum annealing processor," *Phys. Rev. X*, vol. 4, p. 021041, May 2014. URL: `http://link.aps.org/doi/10.1103/PhysRevX.4.021041`.

[2] A. Cho, "Quantum or not, controversial computer yields no speedup," *Science*, vol. 344, no. 6190, pp. 1330–1331, 2014. URL: `http://science.sciencemag.org/content/344/6190/1330`.

[3] D. J. Bernstein, *Post-Quantum Cryptography*, ch. Introduction to post-quantum cryptography, pp. 1–14. Springer Berlin Heidelberg, 2009.

[4] P. W. Shor, "Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer," *SIAM J. Comput.*, vol. 26, pp. 1484–1509, Oct. 1997. URL: `http://dx.doi.org/10.1137/S0097539795293172`.

[5] National Institute of Standards and Technology. "Workshop on cybersecurity in a post-quantum world," [online]. [accessed 2016-07-12]. URL: `http://www.nist.gov/itl/csd/ct/post-quantum-crypto-workshop-2015.cfm`.

[6] V. Lyubashevsky, C. Peikert, and O. Regev, *Advances in Cryptology – EUROCRYPT 2010: 29th Annual International Conference on the Theory and Applications of Cryptographic Techniques, French Riviera, May 30 – June 3, 2010. Proceedings*, ch. On Ideal Lattices and Learning with Errors over Rings, pp. 1–23. Springer Berlin Heidelberg, 2010.

[7] J. W. Bos, C. Costello, M. Naehrig, and D. Stebila, "Post-quantum key exchange for the TLS protocol from the ring learning with errors problem," in *2015 IEEE Symposium on Security and Privacy*, pp. 553–570, May 2015.

[8] E. Alkim, L. Ducas, T. Pöppelmann, and P. Schwabe, "Post-quantum key exchange—a new hope," in *25th USENIX Security Symposium (USENIX Security 16)*, (Austin, TX), pp. 327–343, USENIX Association, 2016. URL: https://cryptojedi.org/papers/newhope-20160328.pdf.

[9] H. Nussbaumer, "Fast polynomial transform algorithms for digital convolution," *IEEE Transactions on Acoustics, Speech, and Signal Processing*, vol. 28, pp. 205–215, Apr 1980.

[10] D. E. Knuth, *The Art of Computer Programming, Volume 2 (3rd Ed.): Seminumerical Algorithms*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1997.

[11] "Unlicense yourself: Set your code free," [online]. [accessed 2016-07-12]. URL: http://unlicense.org.

[12] R. Crandall and C. Pomerance, *Prime Numbers: A Computational Perspective*. Springer-Verlag New York, 2 ed., 2005.

[13] "Cross reference: nussbaumer_mul.c," [online]. [accessed 2016-07-12]. URL: http://code.metager.de/source/xref/gnu/gmp/mpn/generic/nussbaumer_mul.c.

[14] "wbhart/mpir-fft: Bsd licensed fft for mpir," [online]. [accessed 2016-07-12]. URL: https://github.com/wbhart/mpir-fft.

[15] "nussbaumer.c," [online]. [accessed 2016-07-12]. URL: http://web.mit.edu/sage/export/libzn-poly-0.8/nussbaumer.c.

[16] "dstebila/rlwekex: Post-quantum key exchange from the ring learning with errors problem," [online]. [accessed 2016-07-12]. URL: https://github.com/dstebila/rlwekex.

[17] J. W. Bos, M. Naehrig, and J. van de Pol, "Sieving for shortest vectors in ideal lattices: a practical perspective." Cryptology ePrint Archive, Report 2014/880, 2014. http://eprint.iacr.org/2014/880.

[18] "Parallelgausssieve-1.0.tgz," [online]. [accessed 2016-07-12]. URL: http://joppebos.com/src/ParallelGaussSieve-1.0.tgz.

[19] E. Chu and A. George, *Inside the FFT Black Box*. CRC Press, 2016/03/16 1999.

[20] A. Fog. "Instruction tables," [online]. [accessed 2016-05-30]. URL: http://www.agner.org/optimize/instruction_tables.pdf.

[21] W. M. Gentleman and G. Sande, "Fast fourier transforms: For fun and profit," in *Proceedings of the November 7-10, 1966, Fall Joint Computer Conference*, AFIPS '66 (Fall), (New York, NY, USA), pp. 563–578, ACM, 1966.

[22] R. Agarwal and J. Cooley, "New algorithms for digital convolution," in *Acoustics, Speech, and Signal Processing, IEEE International Conference on ICASSP '77.*, vol. 2, pp. 360–362, May 1977.

[23] P. L. Montgomery, "Modular multiplication without trial division," *Mathematics of Computation*, vol. 44, pp. 519–521, 1985.

[24] "Intel® advanced vector extensions programming reference (319433-011)," [online]. [accessed 2016-06-07]. URL: `https://software.intel.com/sites/default/files/m/f/7/c/36945`.

[25] R. T. Moenck, "Practical fast polynomial multiplication," in *Proceedings of the Third ACM Symposium on Symbolic and Algebraic Computation*, SYMSAC '76, (New York, NY, USA), pp. 136–148, ACM, 1976. URL: `http://doi.acm.org/10.1145/800205.806332`.

Appendices

Appendix A

Proofs

Theorem 1. *Let* $\mathbf{R}$ *be a ring with* ω *an* n*-th principal root of unity in* $\mathbf{R}$*, and let* $A = (a_i \in \mathbf{R})_{i=0}^{n-1}$ *and* $B = (b_i \in \mathbf{R})_{i=0}^{n-1}$ *be sequences. Let* $F : \mathbf{R}^n \to \mathbf{R}^n$ *denote the discrete Fourier transformation (using* ω*). Then*

$$\forall k \in \{0, \ldots, n-1\} : F_k(A * B) = F_k(A)F_k(B),$$

where $*$ *denotes the cyclic convolution of the given sequences, and* F_k *indicates the* k*-th component function of* F*.*

Proof. Let $(\tilde{z}_i)_{i=0}^{2n-2}$ be the acyclic convolution of A and B, padded with $\tilde{z}_{2n-1} = 0$, and let $k \in \{0, \ldots, n-1\}$. Then it follows from $\omega^n = 1$ that

$$\begin{aligned}
F_k(A)F_k(B) &= (\sum_{i=0}^{n-1} a_n\omega^{ik})(\sum_{i=0}^{n-1} b_n\omega^{ik}) = \sum_{i=0}^{2n-2}(\sum_{j_1+j_2=i} a_{j_1}b_{j_2})\omega^{ik} \\
&= \sum_{i=0}^{2n-2} \tilde{z}_i\omega^{ik} = \sum_{i=0}^{2n-1} \tilde{z}_i\omega^{ik} = \sum_{i=0}^{n-1} \tilde{z}_i\omega^{ik} + \sum_{i=n}^{2n-1} \tilde{z}_i\omega^{ik} \\
&= \sum_{i=0}^{n-1} \tilde{z}_i\omega^{ik} + \sum_{i=0}^{n-1} \tilde{z}_{i+n}(\omega^n)^k\omega^{ik} = \sum_{i=0}^{n-1}(\tilde{z}_i + \tilde{z}_{i+n})\omega^{ik} \\
&= F_k(A * B).
\end{aligned}$$

So $\forall k \in \{0, \ldots, n-1\} : F_k(A * B) = F_k(A)F_k(B)$.

□

Theorem 2. *Let* $\mathbf{R}$ *be a ring, and* $n, v_1, v_2 \in \mathbb{N}$ *with* $v_1 + v_2 = n$*. Define* $N = 2^n$*,* $m = 2^{v_1}$ *and* $r = 2^{v_2}$*. Then*

$$\mathbf{R}[u]/(u^N + 1) \cong (\mathbf{R}[u_1]/(u_1^r + 1))[u_2]/(u_2^m - u_1)$$

by the mapping:

$$\sum_{i=0}^{N-1} a_i u^i \mapsto \sum_{i=0}^{m-1}\sum_{j=0}^{r-1} a_{mj+i}u_1^j u_2^i.$$

Proof. Call the given mapping Ψ.

If $p \in \mathbf{R}[u]/(u^N+1)$ then there is a unique remainder $q \in \mathbf{R}[u]$ of division of p by u^N+1. Then $p = q$, and q has degree of at most $N-1$. Hence there exists a sequence $(a_i)_{i=0}^{N-1} \in \mathbf{R}^N$ with $q = \sum_{i=0}^{N-1} a_i u^i$, so $\Psi(p)$ is defined. From unicity of q it follows that Ψ is well-defined.

Conversely, if $p \in (\mathbf{R}[u_1]/(u_1^r+1))[u_2]/(u_2^m - u_1)$ then there exists a unique remainder after division of p by $u_2^m - u_1$, $q = \sum_{i=0}^{m-1} A_i u_2^i$ with $q = p$, where $A_i \in \mathbf{R}[u_1]/(u_1^r+1)$. Similarly, each A_i has a unique form $\sum_{j=0}^{r-1} a_{mj+i} u_1^j \equiv A_i \pmod{u_1^r+1}$; this gives us the corresponding a_i such that $\Psi(\sum_{i=0}^{N-1} a_i u^i) = p$, hence Ψ is surjective.

Furthermore, Ψ is trivially injective, as a change in a_i in $\sum_{i=0}^{N-1} a_i u^i$ introduces a similar change for the corresponding a_{mj+i} in the codomain. It follows that Ψ is a bijection.

It is trivial to see that addition is preserved by Ψ and that $\Psi(0) = 0$. It remains to be proven that multiplication is also preserved.

First assume that $p = u$ and $q = cu^k$ with $c \in \mathbf{R}$ and $k \in \mathbb{N}$ with $k < N$. We begin by showing that $\Psi(pq) = \Psi(p)\Psi(q)$.

Find the unique $i \in \{0, \ldots, m-1\}$ and $j \in \{0, \ldots, r-1\}$ such that $mj + i = k$. Then $q = cu^{mj+i}$, so $\Psi(q) = cu_1^j u_2^i$, and it is easily confirmed that $\Psi(p) = u_2$. We find, then, that $\Psi(p)\Psi(q) = cu_1^j u_2^{i+1}$. We distinguish three cases:

1. If $i = m-1$ and $j = r-1$, then $k = mr - 1 = N - 1$, so $\Psi(pq) = \Psi(ucu^k) = \Psi(cu^N) = \Psi(-c) = -c$. On the other hand, $\Psi(p)\Psi(q) = cu_1^{r-1}u_2^m = cu_1^{r-1}u_1 = cu_1^r = -c$. Hence $\Psi(pq) = \Psi(p)\Psi(q)$.

2. If $i = m-1$ but $j \neq r-1$, then $\Psi(pq) = \Psi(cu^{mj+i+1}) = \Psi(cu^{m(j+1)}) = cu_1^{j+1}$. On the other hand, $\Psi(p)\Psi(q) = cu_1^j u_2^m = cu_1^j u_1 = cu_1^{j+1}$. Again, $\Psi(pq) = \Psi(p)\Psi(q)$.

3. If $i \neq m-1$, then $\Psi(pq) = \Psi(cu^{mj+(i+1)}) = cu_1^j u_2^{i+1} = \Psi(p)\Psi(q)$.

So $\Psi(pq) = \Psi(p)\Psi(q)$. It is furthermore easy to see that $\Psi(1q) = \Psi(1)\Psi(q)$. It follows, then, that for $p = u^{k_1}$ and $q = cu^{k_2}$ with $c \in \mathbf{R}$ and $k_1, k_2 \in \mathbb{N}$ with $k_1, k_2 < N$ and $k_1 \geq 1$:

$$\begin{aligned}\Psi(pq) &= \Psi\left([\prod_{i=1}^{k_1} u]cu^{k_2}\right) = \Psi\left(u[\prod_{i=1}^{k_1-1} u]cu^{k_2}\right) \\ &= \Psi(u)\Psi\left([\prod_{i=1}^{k_1-1} u]cu^{k_2}\right).\end{aligned}$$

From induction with as base case $k_1 = 0$ it follows that $\Psi(pq) = \Psi(p)\Psi(q)$.

It is furthermore easy to verify that $\Psi(c_1)\Psi(c_2u^{k_2}) = \Psi(c_1c_2u^{k_2})$, where $c_1, c_2 \in \mathbf{R}$ and $k_1, k_2 \in \mathbb{N}$ with $k_1, k_2 < N$.

Next consider the case where $p = c_1 u^{k_1}$ and $q = c_2 u^{k_2}$ with $c_1, c_2 \in \mathbf{R}$ and $k_1, k_2 \in \mathbb{N}$ with $k_1, k_2 < N$. Then

$$\begin{aligned}\Psi(pq) &= \Psi(c_1 u^{k_1} c_2 u^{k_2}) = \Psi(u^{k_1} c_1 c_2 u^{k_2}) = \Psi(u^{k_1})\Psi(c_1 c_2 u^{k_2}) \\ &= \Psi(u^{k_1})\Psi(c_1)\Psi(c_2 u^{k_2}) = \Psi(c_1)\Psi(u^{k_1})\Psi(c_2 u^{k_2}) \\ &= \Psi(c_1 u^{k_1})\Psi(c_2 u^{k_2}) = \Psi(p)\Psi(q).\end{aligned}$$

Note that the same now holds without the restriction $k_1, k_2 < N$, as there exists a unique polynomial in the equivalence class that is in the form adhering to the restriction, and Ψ is well-defined.

Finally, we consider the general case $p = \sum_{i=0}^{N-1} c_i u^i$ and $q = \sum_{i=0}^{N-1} d_i u^i$, where $c_i, d_i \in \mathbf{R}$.

$$\begin{aligned}\Psi(pq) &= \Psi\left((\sum_{i=0}^{N-1} c_i u^i)(\sum_{i=0}^{N-1} d_i u^i)\right) = \Psi\left(\sum_{i=0}^{N-1}\sum_{j=0}^{N-1}[c_i u^i][d_j u^j]\right) \\ &= \sum_{i=0}^{N-1}\sum_{j=0}^{N-1}\Psi([c_i u^i][d_j u^j]) = \sum_{i=0}^{N-1}\sum_{j=0}^{N-1}\Psi(c_i u^i)\Psi(d_j u^j) \\ &= \left(\sum_{i=0}^{N-1}\Psi(c_i u^i)\right)\left(\sum_{j=0}^{N-1}\Psi(d_j u^j)\right) = \\ &= \Psi\left(\sum_{i=0}^{N-1} c_i u^i\right)\Psi\left(\sum_{i=0}^{N-1} d_i u^i\right) \\ &= \Psi(p)\Psi(q)\end{aligned}$$

Hence multiplication is also preserved, so Ψ is an isomorphism.

□

Theorem 3. *If $m|r$, then $\omega = u^{r/m}$ is a $2m$-th principal root of unity in $\mathbf{R}[u]/(u^r + 1)$.*

Proof. Note that:

$$\omega^{2m} \equiv u^{2r} \equiv -u^r \equiv 1 \pmod{u^r + 1}$$

So ω is indeed a $2m$-th root of unity.

Let $k \in \mathbb{N}$ with $1 \leq k < 2m$. Find $k', l \in \mathbb{N}$ such that $k'l = k$ with k' odd and l a power of two. Define $n = \frac{2m}{l}$; note that this is a power of 2, so in particular $\gcd(k', n) = 1$ and as such, k' has order n in the group $(\mathbb{Z}/n\mathbb{Z}, +)$. This indicates the identity

$$\sum_{i=0}^{n-1}\omega^{ki} = \sum_{i=0}^{n-1}(\omega^l)^{k'i} = \sum_{i=0}^{n-1}(\omega^l)^i$$

the latter equals sign giving a permutation of the order of the (finite) sum. It follows, then, that

$$\begin{aligned}\sum_{i=0}^{n-1}\omega^{ki} &= \sum_{i=0}^{\frac{n}{2}-1}(\omega^l)^i + \sum_{i=\frac{n}{2}}^{n-1}(\omega^l)^i = \sum_{i=0}^{\frac{n}{2}-1}(\omega^{li}+\omega^{li+l\frac{n}{2}})\\ &= \sum_{i=0}^{\frac{n}{2}-1}(\omega^{li}+\omega^{li}\omega^{l\frac{n}{2}}) = \sum_{i=0}^{\frac{n}{2}-1}(\omega^{li}+\omega^{li}\omega^{m})\\ &= \sum_{i=0}^{\frac{n}{2}-1}(\omega^{li}+\omega^{li}u^r) = \sum_{i=0}^{\frac{n}{2}-1}(\omega^{li}+\omega^{li}(-1))\\ &= \sum_{i=0}^{\frac{n}{2}-1} 0 = 0.\end{aligned}$$

Note that if $j = i + tn$ for some $t \in \mathbb{N}$, then

$$\begin{aligned}\omega^{kj} &\equiv \omega^{ki}(\omega^{kn})^t \equiv \omega^{ki}(\omega^{ln})^{k't} \equiv \omega^{ki}(\omega^{2m})^{k't}\\ &\equiv \omega^{ki}1^{k't} \equiv \omega^{ki} \pmod{u^r+1},\end{aligned}$$

and hence:

$$\sum_{i=tn}^{tn+n-1}\omega^{ki} \equiv 0 \pmod{u^r+1}.$$

As such

$$\sum_{i=0}^{2m}\omega^{ki} \equiv \sum_{t=0}^{l-1}\sum_{i=tn}^{tn+n-1}\omega^{ki} \equiv \sum_{t=0}^{l-1} 0 \equiv 0 \pmod{u^r+1},$$

which holds for all $1 \leq k < 2m$, such that ω is also a principal $2m$-th root of unity.

□

Theorem 4. *Let $A = (a_{i,j})$ and $B = (b_{i,j})$ be $2m \times m$ matrices in $\mathbf{R}[u]/(u^r+1)$, and let $C = (c_{i,j})$ an $m \times 2m$ matrix in the same ring. Let $\omega = u^{r/m}$ be a $2m$-th principle root of unity. Say for $i, k, l < m$*

$$S_{i,k,l} = \sum_{j=0}^{2m-1} c_{i,j}a_{j,k}b_{j,l} = \begin{cases}1 & \text{if } k+l=i,\\ u & \text{if } k+l=i+m,\\ 0 & \text{otherwise.}\end{cases}$$

Define two matrices $C' = (c'_{i,j})$ and $A' = (a'_{i,j})$ with

$$c'_{i,j} = \begin{cases}a_{j,0} & \text{if } i=0,\\ -u^{r-1}a_{j,m-i} & \text{otherwise}\end{cases}$$

and

$$a'_{i,j} = \begin{cases} c_{0,i} & \text{if } j = 0, \\ uc_{m-j,i} & \text{otherwise,} \end{cases}$$

and let

$$S'_{i,k,l} = \sum_{j=0}^{2m-1} c'_{i,j} a'_{j,k} b_{j,l}.$$

Then $S_{i,k,l} = S'_{i,k,l}$ *for* $i, k, l \in \{0, 1, \ldots, m-1\}$.

Proof. Let $i, k, l \in \{0, 1, \ldots, m-1\}$.

Note that

$$(m-i) + l = (m-k) \iff k + l = i, \text{ and}$$

$$(m-i) + l = (m-k) + m \iff k + l = i + m,$$

so $S_{m-k,m-i,l} = S_{i,k,l}$. We see, then, that if $i \neq 0$ and $k \neq 0$,

$$\begin{aligned} S'_{i,k,l} &= \sum_{j=0}^{2m-1} c'_{i,j} a'_{j,k} b_{j,l} = \sum_{j=0}^{2m-1} -u^r c_{m-k,j} a_{j,m-i} b_{j,l} \\ &= \sum_{j=0}^{2m-1} -\omega^m c_{m-k,j} a_{j,m-i} b_{j,l} = \sum_{j=0}^{2m-1} c_{m-k,j} a_{j,m-i} b_{j,l} \\ &= S_{m-k,m-i,l} = S_{i,k,l}. \end{aligned}$$

If $i = 0$ and $k \neq 0$, note that $k + l = i$ can not hold, so

$$S_{i,k,l} = \begin{cases} u & \text{if } k + l = m, \\ 0 & \text{otherwise,} \end{cases}$$

and similarly $0 + l = (m-k) + m$ can never hold, such that

$$S'_{m-k,0,l} = \begin{cases} 1 & \text{if } 0 + l = m - k, \\ 0 & \text{otherwise,} \end{cases}$$

which makes it easy to see that $S_{0,k,l} = uS_{m-k,0,l}$. Hence

$$\begin{aligned} S'_{0,k,l} &= \sum_{j=0}^{2m-1} c'_{0,j} a'_{j,k} b_{j,l} = \sum_{j=0}^{2m-1} uc_{m-k,j} a_{j,0} b_{j,l} \\ &= uS_{m-k,0,l} = S_{0,k,l}. \end{aligned}$$

Similarly if $i \neq 0$ and $k = 0$, $k+l = i+m$ can not hold and $(m-k)+l = 0$ can not hold, hence

$$S_{i,k,l} = \begin{cases} 1 & \text{if } k + l = i, \\ 0 & \text{otherwise,} \end{cases}$$

and

$$S'_{0,m-i,l} = \begin{cases} u & \text{if } (m - i) + l = m, \\ 0 & \text{otherwise.} \end{cases}$$

As $k + l = i \iff (m - i) + l = m$, $uS_{i,0,l} = S_{0,m-i,l}$, and we find that

$$\begin{aligned} S'_{i,0,l} = \sum_{j=0}^{2m-1} c'_{i,j} a'_{j,0} b_{j,l} = \sum_{j=0}^{2m-1} -u^{r-1} c_{0,j} a_{j,m-i} b_{j,l} = \\ = -u^{r-1} S_{0,m-i,l} = -u^r S_{i,0,l} = S_{i,0,l}. \end{aligned}$$

Finally, if $i = 0$ and $k = 0$, then

$$S'_{0,0,l} = \sum_{j=0}^{2m-1} c'_{0,j} a'_{j,0} b_{j,l} = \sum_{j=0}^{2m-1} c_{0,j} a_{j,0} b_{j,l} = S_{0,0,l}.$$

We conclude that for any $i, k, l \in \{0, 1, \ldots, m-1\}$: $S'_{i,k,l} = S_{i,k,l}$. $\square$

Appendix B

Nussbaumer Source Code

```
/**
 * @file NegaNussbaumer.h
 * @author Gerben van der Lubbe
 *
 * File containing Nussbaumer's negacyclic convolution algorithm.
 */

#ifndef NEGANUSSBAUMER_H
#define NEGANUSSBAUMER_H

#include <vector>
#include <cassert>
#include <cmath>

#include "Polynomial.h"
#include "BitManip.h"

/**
 * Class for performing the Negacyclic Nussbaumer algorithm.
 * To run it, transform both polynomials, one with the slow
 * transform and one with the fast transform; perform a
 * componentwise() product, and calculate the inverse
 * transform.
 */
template<typename RingElt>
class NegaNussbaumer {
public:
  /// Transformed polynomial
  typedef std::vector<Polynomial<RingElt>> Transformed;

  NegaNussbaumer(std::size_t N);

  Transformed         transformSlow(
                          const Polynomial<RingElt>& orig,
                          bool fixFactor = true
                      ) const;
  Transformed         transformFast(
```

```
                        const Polynomial<RingElt>& orig
                    ) const;
  Polynomial<RingElt> inverseTransform(
                        const Transformed& trans
                    ) const;

  Transformed               componentwise(
                                const Transformed& t1,
                                const Transformed& t2
                            ) const;
  static Polynomial<RingElt> multiply(
                                std::size_t N,
                                const Polynomial<RingElt>& p1,
                                const Polynomial<RingElt>& p2
                            );

protected:
  Polynomial<RingElt> addRotatedPolynomial(
                        const Polynomial<RingElt>& p1,
                        const Polynomial<RingElt>& p2,
                        int steps
                    ) const;
  Polynomial<RingElt> rotatePolynomial(
                        const Polynomial<RingElt>& pol,
                        int steps
                    ) const;

  Polynomial<RingElt> correct(
                        const Polynomial<RingElt>& p
                    ) const;
  unsigned int        getFactor() const;

private:
  std::size_t n_;
  std::size_t m_, r_;
};


/**
 * Constructor for an object that will perform
 * multiplications on the given polynomial modulo u^N + 1,
 * according to the Nussbaumer algorithm. The value "N"
 * must be a power of 2 for this algorithm.
 * @param[in] N  The "N" of the algorithm; the
 *               multiplication is calculated modulo
 *               "u^N + 1". This must be greater than 2, as
 *               a trivial alternative should be used there.
 */
template<typename RingElt>
NegaNussbaumer<RingElt>::NegaNussbaumer(
    std::size_t N
) {
  assert(N > 1);
```

```
  // Get the n = log_2 N (which must be an integer)
  n_ = 0;
  while((1u << n_) < N)
    ++n_;
  assert((1u << n_) == N);

  // Find m = 2^lg_m and r = 2^lg_m (with lg_m and lg_r
  // integers), such that m*r = n with r minimum; that is,
  // m = floor(lg_n/2) and lg_m + lg_r = n.
  std::size_t lg_m, lg_r;
  lg_m = n_ >> 1;
  lg_r = n_ - lg_m;
  m_ = 1 << lg_m;
  r_ = 1 << lg_r;
}


/**
 * Perform the full multiplication of the two given
 * polynomials, modulo u^N + 1.
 * @param[in] N    The N in the modulo u^N + 1
 * @param[in] p1   The first polynomial to multiply.
 * @param[in] p2   The second polynomial to multiply.
 * @return    The Negacyclic convolution
 */
template<typename RingElt>
Polynomial<RingElt> NegaNussbaumer<RingElt>::multiply(
                        std::size_t N,
                        const Polynomial<RingElt>& p1,
                        const Polynomial<RingElt>& p2
                    ) {
  // Otherwise, recurse into the algorithm again
  NegaNussbaumer<RingElt> nussbaumer(N);
  auto t1 = nussbaumer.transformSlow(p1, false);
  auto t2 = nussbaumer.transformFast(p2);
  auto resTrans = nussbaumer.componentwise(t1, t2);
  return nussbaumer.inverseTransform(resTrans);
}


/**
 * Perform the componentwise multiplication of the
 * transformed polynomials. One must be transformed through
 * the transformSlow method, the other through the
 * transformFast method.
 * @param[in] slow   The slow-transformed polynomial.
 * @param[in] fast   The fast-transformed polynomial.
 * @return The transformed result of the multiplication.
 */
template<typename RingElt>
typename NegaNussbaumer<RingElt>::Transformed
                  NegaNussbaumer<RingElt>::componentwise(
                      const Transformed& slow,
                      const Transformed& fast
```

```
                ) const {
  // Special case where N = 2, where Nussbaumer's algorithm
  // is not applicable.
  if(n_ == 1) {
    Transformed resTrans;
    Polynomial<RingElt> res(2);
    RingElt t = slow[0][0]*fast[0][2];
    res[0] = t - slow[0][1]*fast[0][1];
    res[1] = t + slow[0][2]*fast[0][0];
    resTrans.push_back(res);
    return resTrans;
  }

  Transformed resTrans;
  resTrans.push_back(Polynomial<RingElt>(r_));
  for(std::size_t i = 1; i < slow.size(); ++i) {
    auto term = NegaNussbaumer<RingElt>::multiply(
                                r_, slow[i], fast[i]
              );
    resTrans.push_back(term);
  }

  return resTrans;
}


/**
 * Transform the polynomial to the list of polynomials
 * that can be multiplied componentwise (see algorithm
 * description for a more thorough explanation). This is
 * the slow transform; the other polynomial input for
 * "componentwise" must be a polynomial transformed by
 * transformFast.
 * @param[in] orig      The original polynomial, must be of
 *                      degree N.
 * @param[in] fixFactor Whether to compensate for the
 *                      factor (should be false for
 *                      recursive calls).
 * @return    The transformed polynomial.
 */
template<typename RingElt>
typename NegaNussbaumer<RingElt>::Transformed
             NegaNussbaumer<RingElt>::transformSlow(
                 const Polynomial<RingElt>& orig,
                 bool fixFactor
             ) const {
  assert(orig.getSize() == (1u << n_));

  // Handle the special case N = 2, where another algorithm
  // is used (with some pre-processing)
  // We make 2 as the first componentwise multiplication is
  // skipped.
  if(n_ == 1) {
    Transformed trans(1, Polynomial<RingElt>(3));
```

```
  trans[0][0] = orig[0];
  trans[0][1] = orig[0] + orig[1];
  trans[0][2] = orig[1] - orig[0];
  return trans;
}

Transformed trans(2*m_, Polynomial<RingElt>(r_));

// Correct the scale of the polynomial. Do so now, as
// this needs less steps then at a later point. Also, as
// this result may be re-used, it's better to do here
// than at the end.
Polynomial<RingElt> scaledOrig =
                      fixFactor ? correct(orig) : orig;

// Get the input polynomials, transformed, applying the
//  C_4' matrix immediately.
for(std::size_t j = 0; j < r_; ++j)
  trans[0][j] = scaledOrig[m_*j];
for(std::size_t i = 1; i < m_; ++i)
  trans[i][0] = -scaledOrig[m_*(r_  - 1) + m_ - i];
for(std::size_t i = 1; i < m_; ++i) {
  for(std::size_t j = 1; j < r_; ++j) {
    trans[i][j] = scaledOrig[m_*(j - 1) + m_ - i];
  }
}

// Apply the C_3^T matrix.
for(std::size_t i = 0; i < m_ - 1; ++i)
  trans[m_ + i][0] = -trans[i][r_ - 1];
for(std::size_t i = 0; i < m_ - 1; ++i) {
  for(std::size_t j = 1; j < r_; ++j) {
    trans[m_ + i][j] = trans[i][j - 1];
  }
}

// Apply the C_2^T matrix.
Polynomial<RingElt> lastEntry(-trans[0]);
for(std::size_t i = 1; i < 2*m_ - 1; ++i)
  lastEntry -= trans[i];

trans[2*m_ - 1] = lastEntry;

// Perform the FFT
std::size_t j = (n_ >> 1) + 1;
while(j > 0) {
  --j;

  for(std::size_t sPart = 0;
      sPart < (m_ >> j);
      ++sPart) {
    std::size_t s, sRev;
    s = sPart << (j+1);
    sRev = bitrev((n_ >> 1) - j, sPart) << j;
```

```
      int k = -static_cast<int>((r_/m_)*sRev);

      for(std::size_t t = 0; t < (1u << j); ++t) {
        std::size_t e, f;
        e = s + t;
        f = e + (1u << j);

        // Now, we set (simultaneously):
        // trans[e] = trans[e] + u^k*trans[f]
        // trans[f] = trans[e] - u^k*trans[f]
        Polynomial<RingElt> tmp;
        if(e == 0 && j == 0) {
          // Don't calculate trans[0]; we don't need it.
          tmp = Polynomial<RingElt>(r_);
        }
        else {
          tmp = addRotatedPolynomial(
                    trans[e], trans[f], k
                );
        }
        trans[f] = addRotatedPolynomial(
                        trans[e], trans[f], k + r_
                    );
        trans[e] = tmp;
      }
    }
  }

  return trans;
}


/**
 * Transform the polynomial to the list of polynomials that
 * can be multiplied componentwise (see algorithm
 * description for a more thorough explanation). This is
 * the fast transform; the other polynomial input for
 * "componentwise" must be a polynomial transformed by
 * transformSlow.
 * @param[in] orig     The original polynomial, must be of
 *                     degree N.
 * @return    The transformed polynomial.
 */
template<typename RingElt>
typename NegaNussbaumer<RingElt>::Transformed
             NegaNussbaumer<RingElt>::transformFast(
                 const Polynomial<RingElt>& orig
             ) const {
  assert(orig.getSize() == (1u << n_));

  // Prepare for another algorithm if N = 2.
  if(n_ == 1) {
    Transformed trans(1, Polynomial<RingElt>(3));
```

```
  trans[0][0] = orig[0];
  trans[0][1] = orig[1];
  trans[0][2] = orig[0] + orig[1];
  return trans;
}


Transformed trans(2*m_, Polynomial<RingElt>(r_));

// First get the polynomials to perform the fourier
// transform on. These are 2m polynomials of which r
// coefficients will be considered, where two sets of
// m polynomials are created by shuffling "orig".
for(std::size_t i = 0; i < 2*m_; ++i) {
  for(std::size_t j = 0; j < r_; ++j) {
    trans[i][j] = orig[m_*j + (i % m_)];
  }
}

// Do the fast fourier transform.
std::size_t j = (n_ >> 1);
while(j > 0) {
  --j;

  for(std::size_t sPart = 0;
      sPart < (m_ >> j);
      ++sPart) {
    std::size_t s, sRev;
    s = sPart << (j+1);
    sRev = bitrev((n_ >> 1) - j, sPart) << j;

    int k = static_cast<int>((r_/m_)*sRev);

    for(std::size_t t = 0; t < (1u << j); ++t) {
      std::size_t e, f;
      e = s + t;
      f = e + (1u << j);

      // Now, we set (simultaneously):
      // trans[e] = trans[e] + u^k*trans[f]
      // trans[f] = trans[e] - u^k*trans[f]
      Polynomial<RingElt> tmp;
      if(e == 0 && j == 0) {
        // Don't calculate trans[0]; we don't need it.
        tmp = Polynomial<RingElt>(r_);
      }
      else {
        tmp = addRotatedPolynomial(
                  trans[e], trans[f], k
              );
      }
      trans[f] = addRotatedPolynomial(
                     trans[e], trans[f], k + r_
                 );
```

```
        trans[e] = tmp;
      }
    }
  }

  return trans;
}


/**
 * Perform the inverse transform (see paper).
 * @param[in] trans    The transformed form of the
 *                     polynomial.
 * @return    The polynomial form.
 */
template<typename RingElt>
Polynomial<RingElt>
    NegaNussbaumer<RingElt>::inverseTransform(
        const Transformed& trans
    ) const {
  // Special case for N = 2, where no actual inverse
  // transform is needed
  if(n_ == 1)
    return trans[0];

  // Do the inverse FFT (through a DIT with
  // unordered input)
  Transformed z(trans);
  std::size_t jMax = n_ >> 1;
  for(std::size_t j = 0; j <= jMax; ++j) {
    for(std::size_t t = 0; t < (1u << j); ++t) {
      int k = (int)((r_/m_)*( t << (jMax - j) ));

      for(std::size_t s = 0; s < 2*m_; s += (1 << (j+1))) {
        std::size_t e, f;
        e = s + t;
        f = e + (1u << j);

        Polynomial<RingElt> tmp;
        if(j == 0 && e == 0) {
          z[e] = z[f];
          z[f] = -z[f];
        }
        else {
          // Now, we set (simultaneously):
          // z[e] = z[e] + u^k*z[f]
          // z[f] = z[e] - u^k*z[f]
          tmp = addRotatedPolynomial(z[e], z[f], k);
          if(j != jMax) {
            // We don't need the last half of the result.
            z[f] = addRotatedPolynomial(
                       z[e], z[f], k + r_
                   );
          }
```

```
          z[e] = tmp;
        }
      }
    }
  }

  // Inverse the order of all but the first element.
  std::size_t from, to;
  for(from = 1, to = m_ - 1; from < to; from++, to--) {
    std::swap(z[from], z[to]);
  }

  // Multiply all but the first element with
  // -u^{r-1} = u^{2r - 1}
  for(std::size_t i = 1; i < m_; ++i)
    z[i] = rotatePolynomial(z[i], 2*r_ - 1);

  // Unpack the polynomial
  Polynomial<RingElt> res(1u << n_);
  for(std::size_t i = 0; i < m_; ++i) {
    for(std::size_t j = 0; j < r_; ++j) {
      res[m_*j + i] = z[i][j];
    }
  }

  return res;
}


/**
 * Calculate p1 + (u^steps)*p2 modulo u^r + 1. This could
 * be done by a separate rotate/add step, but this would
 * require possible negations followed by additions,
 * rather than immediately subtracting.
 * @param[in] p1      The first polynomial
 * @param[in] p2      The second polynomial.
 * @param[in] steps   The number of steps to "rotate" p2.
 * @return    p1 + (u^steps)*p2 modulo u^r + 1.
 */
template<typename RingElt>
Polynomial<RingElt>
    NegaNussbaumer<RingElt>::addRotatedPolynomial(
        const Polynomial<RingElt>& p1,
        const Polynomial<RingElt>& p2,
        int steps
    ) const {
  Polynomial<RingElt> ret(r_);

  // Get "steps" as negative value, with 2*r_ < steps <= 0
  steps %= 2*r_;
  if(steps > 0)
    steps -= 2*r_;

  for(std::size_t i = 0; i < r_; ++i) {
```

```
    int src = static_cast<int>(i) - steps;
    std::size_t srcIndex = static_cast<size_t>(src) % r_;
    bool signInverse = (src / r_) & 1;

    if(signInverse)
      ret[i] = p1[i] - p2[srcIndex];
    else
      ret[i] = p1[i] + p2[srcIndex];
  }

  return ret;
}


/**
 * The function "rotates" the polynomial, as though
 * multiplying it with u^steps, modulo u^r + 1.
 * @param[in] pol    The polynomial to rotate.
 * @param[in] steps  The number of steps to rotate.
 * @return    The resulting polynomial.
 */
template<typename RingElt>
Polynomial<RingElt>
    NegaNussbaumer<RingElt>::rotatePolynomial(
        const Polynomial<RingElt>& pol,
        int steps
    ) const {
  Polynomial<RingElt> ret(r_);

  // Get "steps" as negative value, with 2*r_ < steps <= 0
  steps %= 2*r_;
  if(steps > 0)
    steps -= 2*r_;

  for(std::size_t i = 0; i < r_; ++i) {
    int src = static_cast<int>(i) - steps;
    std::size_t srcIndex = static_cast<size_t>(src) % r_;
    bool signInverse = (src / r_) & 1;

    if(signInverse)
      ret[i] = -pol[srcIndex];
    else
      ret[i] = pol[srcIndex];
  }

  return ret;
}


/*
 * Corrects the result of the Nussbaumer algorithm by
 * dividing the factor of getFactor out of this one.
 * @param[in] p    The polynomial to correct.
 * @return    The polynomial.
```

```
 */
template<typename RingElt>
Polynomial<RingElt>
    NegaNussbaumer<RingElt>::correct(
        const Polynomial<RingElt>& p
    ) const {
  // To calculate the inverse FFT we need the inverse of
  // the correction factor
  RingElt inverseElt;
  int inverse;
  if(!RingElt::getInverse(inverseElt, getFactor())) {
    std::cerr << "Factor does not have an inverse"
                 " in the given ring" << std::endl;
    exit(1);
  }

  inverse = inverseElt.toInt();

  // Multiply with the inverse
  auto ret = p;
  ret *= inverse;
  return ret;
}


/**
 * Calculate the factor that the result is multiplied with
 * after the Nussbaumer algorithm. The final step should be
 * to multiply the result with the inverse of this factor.
 * @return    The factor.
 */
template<typename RingElt>
unsigned int NegaNussbaumer<RingElt>::getFactor() const {
  unsigned int factor = 1;
  unsigned int nTest = n_;
  while(nTest != 1) {
    unsigned int lgmTest = nTest >> 1;
    unsigned int lgrTest = nTest - lgmTest;
    factor *= 2*(1 << lgmTest);
    nTest = lgrTest;
  }
  return factor;
}

#endif
```