國立臺灣大學電機資訊學院資訊工程學研究所

碩士論文

Department of Computer Science & Information Engineering

College of Electrical Engineering and Computer Science

National Taiwan University

Master's Thesis

多項式乘法的新技巧：
驗證應用單項式於中國剩餘定理之多項式乘法

A New Trick for Polynomial Multiplication:
A verified CRT polymul utilizing a monomial factor

邱俊茗

Chun-Ming Chiu

指導教授: 蕭旭君 博士

Advisor: Hsu-Chun Hsiao, Ph.D.

中華民國 114 年 7 月

July, 2025

doi:10.6342/NTU202501941

致謝

非常感謝王柏堯教授在這兩年以上的期間，願意以共同指導教授的身份，盡心盡力的帶領我進行研究。另外也非常感謝楊柏因教授用心的教學，讓我能夠順利入門密碼學實作與優化技巧的研究。最後想要感謝蕭旭君教授與呂學一教授提供的協助，讓我可以在碩士期間能夠專心於課程學習與研究進行。

doi:10.6342/NTU202501941

doi:10.6342/NTU202501941

摘要

針對多項式乘法的問題，我們在論文中展示了新穎的轉換策略，並示範如何將其應用在 NTRU Prime 密碼系統上。明確而言，在此密碼系統的所有參數集中，我們特別關注了 `sntrup761` 跟 `ntrulpr761` 這兩者。它們使用的多項式商環都是 $\mathbb{Z}_{4591}[x]/\langle x^{761} - x - 1\rangle$。為了評估我們的新想法是否具實用性，我們使用了 C++ 語言與 ARM Neon intrinsics，將新提出的演算法實做出來。過程中我們發現到，代數轉換過程中事實上存在著不少優化契機。我們進一步利用了這些機會，最終設計出了效率十分出色的乘法器，其在 Cortex-A72 的平台上的計算速度勝過了所有現有紀錄。

通常而言，為了避免整數溢流錯誤，基於整數模運算的演算法會經常需要更換計算過程的中間值，轉而用另一個同餘但絕對值較小的整數取代。為了追求效率，我們在實做中盡可能跳過了這個步驟。不可否認的，這增加了整數溢流錯誤的危險性。針對這類型的錯誤，由於引發機率實在過低，基於測試資料的傳統偵測方法往往是沒有幫助的。為了確認我們實做出的是否正確無誤，我們運用了名為 CryptoLine 的形式驗證工具。驗證過程中，我們使用了 CryptoLine 最新版本的所有功能，其中幫助最大的兩個功能是基於 Integer Set Library 的值域驗證，以及程式等價性的驗證。透過後者，我們可以將同一個程式碼用不同的設定，編譯成另一個「優化較不徹底但容易驗證」的執行檔。最終透過證明新版本的正確性以及兩者的等價性，我們可以間接得到原版本執行檔的正確性保證。

doi:10.6342/NTU202501941

關鍵字：多項式乘法、數論轉換、混合底數、中國剩餘定理、整數模運算

doi:10.6342/NTU202501941

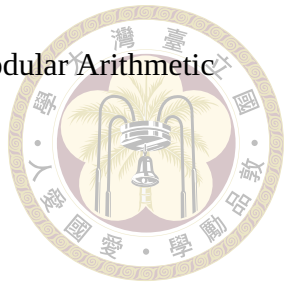
Abstract

In this paper we present a novel transformation strategy for polynomial multiplications and apply it to NTRU Prime, specifically the parameter sets `sntrup761` and `ntrulpr761` working in the ring $\mathbb{Z}_{4591}[x]/\langle x^{761} - x - 1\rangle$. To evaluate the practicality of our idea, we implemented the algorithm in C++ with ARM Neon intrinsics. By further exploiting the various optimization opportunities in the transformation process, we achieve state-of-the-art performance on Cortex-A72.

Because of the aggressively lazy modular reduction strategy, overflows are of serious concern. Such errors in an optimized implementation are notoriously difficult to detect using traditional test vectors. To this end, the compiled binary file is formally verified using the tool CryptoLine. We use all the features in the current version of CryptoLine. This includes the Integer Set Library for range checking, plus the Logical Equivalence Checking to verify the correctness of the binary produced with the most optimized compiler setting by showing it as being equivalent to a binary from a less optimized compilation.

doi:10.6342/NTU202501941

Keywords: Polynomial multiplication, NTT, Mixed-radix, CRT, Modular Arithmetic



doi:10.6342/NTU202501941

Contents

doi:10.6342/NTU202501941

doi:10.6342/NTU202501941

doi:10.6342/NTU202501941

doi:10.6342/NTU202501941

List of Figures

doi:10.6342/NTU202501941

doi:10.6342/NTU202501941

List of Tables

doi:10.6342/NTU202501941

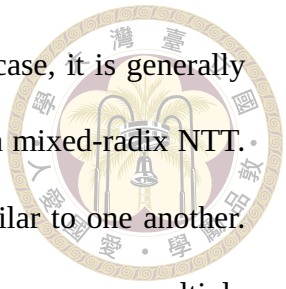
doi:10.6342/NTU202501941

Chapter 1 Introduction

For the past few years, a lot of effort was put into NTT-based polynomial multiplication (polymul) algorithms, as a part of the quest for ever faster implementations for lattice-based cryptoschemes. Thanks to the cumulative effort, the best practice for implementing a standard radix-2 NTT has been figured out for the most part, leaving little room for further improvement. In contrast, the mixed-radix variant has received less attention. This is to be expected, considering that the parameter sets for many well known cryptoschemes (Kyber, Dilithium, etc.) are deliberately picked to ensure that the quotient rings are "radix-2 friendly". Nonetheless, for those cryptoschemes not designed as such, mixed radix is regarded as a competetive alternative to Schönhage/Nussbaumer or Toom-Cook/Karatsuba, depending on the size of the ring. In fact, multiple practical examples have shown that mixed radix is often the best approach to tackle those rings. A recent culminating work in this line is [17], in which Hwang described a mixed-radix polymul for `sntrup761` and reported a staggering 2x speed-up on AVX2 compared to the previous state-of-the-art based on Schönhage/Nussbaumer.

Despite the promising potential, it is often difficult to fully harness the power of mixed-radix NTT. Firstly, the choice of the radix combination is heavily restricted, both by the structure of the coefficient ring and the degree of input polynomial. These restrictions often rule out otherwise attractive combinations, forcing us to make do with choices that

doi:10.6342/NTU202501941



are less efficiently computable. Secondly, compared to the radix-2 case, it is generally harder to optimize the timing and strategy for modular reduction in a mixed-radix NTT. Because of the different radices, each layer is fundementally dissimilar to one another. To make matter worse, for the fixed radix on some specific layer, there are even multiple designs with trade-off between the output range and computational effort. Because of the difficulty in range analysis, oftentimes one would just preferably settle on conservative designs that for sure wouldn't overflow.

We address these two issues in this paper. For the former, we propose the use of Chinese remainder theorem (CRT) with monomial factor to alleviate the degree restriction. This opens up new possibilities in the choice of radix combinations and further enables the use of specific radices that are more optimizable with Rader's trick. We illustrate the effectiveness of this idea by implementing a optimized polymul for `sntrup761` in ARM Neon and showing that it outperforms the current state-of-the-art [17]. For the latter, we demonstrate how CryptoLine can be used to automate the check for the absense of overflows. This encourages the implementer to try out more aggressively lazy reduction strategies without constantly worrying about the risk of overflows. While doing so, we also verified the algebraic techniques used in our implementation, showing simultaneously the correctness of our "CRT with monomial factor" idea and other proposed tricks, and the potential of CryptoLine to aid the design process of exotic polymul techniques.

1.1 Contributions

Our main contributions are summarized as follows.

- We introduce the use of monomial factor $x^{n_{\text{low}}}$ in CRT as a complement to NTT-

doi:10.6342/NTU202501941

based multiplications.

- We discuss how Rader's trick leads to highly optimized radix-3 and radix-5 NTT designs.
- We present a way to exploit the time shifting property of DFT and perform zero-skipping in Good-Thomas NTT with almost no overhead in code size.
- We propose some specialized variants of Barrett multiplication.
- We combine the ideas above and implement a multiplier for `sntrup761` in ARM Neon, outperforming the state-of-the-art in [17] by a considerable margin.
- We formally verify the correctness of our implementation and at the same time showcase CryptoLine's capability of handling complicated modular reduction strategies and exotic algebraic techniques.

1.2 Related Work

NTRU Prime Since NTRU Prime's quotient rings (which are actually Galois fields by design) are naturally not NTT-friendly, many tricks had been used to do fast multiplication. For methods retaining the integer modulus, initially Toom-Cook was used. The change-of-modulus-into-NTT method which was mentioned as early as (among other places) [7] was first introduced to NTRU Prime in the submission [11] (along with the very rare Rader-17, in the ring $\mathbb{Z}_{4591}[x]/\langle x^{1530}-1\rangle$) and lattice-based crypto at large in [1]. While the code package with [8] already contained such code, [9] seems to be the first paper to use $\mathbb{Z}_{4591}[x]/\langle(x^{1024}+1)(x^{512}-1)\rangle$ in an truncated Schönhage. [19] continued to use Rader-17 with $\mathbb{Z}_{4591}[x]/\langle x^{1632}-1\rangle$ (coming down to length-16 convolutions), and

doi:10.6342/NTU202501941

[17] improved this to T(runcated)Rader-17 and the smaller ring $\mathbb{Z}_{4591}[x]/\langle\Phi_{17}(x^{96})\rangle$. For NTT multiplications switching the integral modulus, [1] had an incomplete NTT with $\mathbb{Z}_{6984193}[x]/\langle x^{1536}-1\rangle$. *Despite all these papers, the use of a monomial modulus for CRT multiplication including a multidimensional Good's trick seems to be quite new.*

Other There are many papers over the last few years about polynomial multiplications in Kyber, Dilithium, Saber, and NTRU, the NIST lattice finalists. The most recent advances seems to be [11] introduced the use of Cooley-Tukey butterflies going both for Forward and Inverse NTTs; [23], [6] and [16] introduced signed Montgomery, Barrett and Plantard multiplications respectively. For non-NTT polymul, [10] describes the state of the art (Toom-based TMVP methods). There are a few papers dealing with verified lattice-based crypto code, such as [2, 3, 20], all of them using mostly 2-way NTTs. There are also many articles in the vein of [5], but those are protocol-level verification. Verification of an "non-standard" CRT-NTT multiplication is new as far as we are able to determine, and so is the use of rightshifts replacing divisions including their associated range verifications.

doi:10.6342/NTU202501941

Chapter 2 Preliminaries

In the following, it is understood that $q = 4591$, and coefficients are all from $\mathbb{Z}_q = \mathbb{Z}/q\mathbb{Z}$, unless explictly stated otherwise. Sufficient to uniquely designate each equivalence class modulo some degree-d polynomial $Q(x)$, the polynomials $P(x)$ with degree at most $d-1$ are often considered together. By storing the degree-i coefficient as `P[i]` within an array `P` of length d (with trailing zeros sometimes required), the set can be identified with the vector space $\mathbb{Z}_q^d$ (or the free module R^d if the coefficients are from some other ring R). To ease the discussion in the following, we will abuse language slightly and use "polynomials of length d" to describe polynomial with degree at most $d-1$.

2.1 NTRU Prime

The NTRU Prime family comprises an NTRU variant PKE (public-key encryption) and a Ring-Learning-with-Rounding variant PKE, using ternary errors and a finite field for a polynomial ring, which are made into two KEMs (Key Establishment Methods) via a Fujisaki-Okamoto Transform [13]. There are a variety of possible parametrizations, but the main recommended variants uses the finite field $\mathbb{Z}_{4591}[x]/\langle x^{761} - x - 1)\rangle \cong \mathbb{F}_{4591^{761}}$ as the polynomial ring. We refer the reader to [8] for details. It suffices to mention that *the main operation during encapsulation and decapsulation is multiplying a random-looking*

doi:10.6342/NTU202501941

polynomial with a ternary polynomial.

2.2 Modular Reductions and Multiplications

In the context of lattice-based crypto, modular reductions means to select a smaller-in-magnitude congruential representative. Often the reductions are "non-canonical", in the sense that the canonical representative modulo q—whether the standard/unsigned one (inside $[0, q-1]$) or the centered/signed one (inside $[\frac{-q+1}{2}, \frac{q-1}{2}]$, assuming q odd)—is not necessarily obtained. The objective is simply to reach a integer small enough in magnitude that arithmetic can continue to happen without overflowing. A modular multiplication (mulmod) is to find a small enough congruential representative of a product, which again need not be canonical. Currently there are three standard ways to perform modular multiplications: Barrett [6], Montgomery [23] and Plantard [16]. In our implementation, we predominately use Barrett multiplication (and some novel variant of it, discussed later in Section 3.4.2).

2.2.1 Barrett Reduction and Multiplication

We say that the function $[\![\cdot]\!] : \mathbb{R} \to \mathbb{Z}$ is an *integer approximation* if $|x - [\![x]\!]| \leq 1$, and define the associated modulo operator as $a \bmod^{[\![\cdot]\!]} m = a - [\![a/m]\!]m$. For example, the floor function $\lfloor \cdot \rfloor$ clearly qualifies as an integer approximation, and $a \bmod^{\lfloor \cdot \rfloor} m \in [0, m-1]$ gives back the standard modulo operator $a \bmod m$. Note that the assumption of integer approximation gives $|a/m - [\![a/m]\!]| \leq 1$, which implies $|a \bmod^{[\![\cdot]\!]} m| = m|a/m - [\![a/m]\!]| \leq m$.

When $[\![\cdot]\!]_0$, $[\![\cdot]\!]_1$ are integer approximations, a representative of $ab \bmod q$ is available

doi:10.6342/NTU202501941

via $ab - eq \equiv ab \pmod{q}$, where $e = \left[\!\left[\frac{a\left[\!\left[\frac{bR}{q} \right]\!\right]_0}{R} \right]\!\right]_1 \approx \frac{ab}{q}$. Becker et al. [6] showed[1] that

$$|ab - eq| \leq \frac{1}{R}\left[|a| \left| \mathrm{mod}^{[\![\cdot]\!]_0} q \right| + \left| \mathrm{mod}^{[\![\cdot]\!]_1} R \right| q \right], \tag{2.1}$$

where $\left|\mathrm{mod}^{[\![\cdot]\!]} N\right|$ denotes $\max_a \left| a \bmod^{[\![\cdot]\!]} N \right|$ for the integer approximation $[\![\cdot]\!]$. They also noted that for fixed b, one can and should precompute $\bar{b} = [\![bR/q]\!]_0$. For implementation in Neon-like instruction sets, the 16-bit `sqrdmulh` computes $\lfloor 2a\bar{b}/2^{16} \rceil = \lfloor a\bar{b}/2^{15} \rceil$, so the most natural choice is $R = 2^{15}$ and $[\![\cdot]\!]_0 = [\![\cdot]\!]_1 = \lfloor \cdot \rceil$, with precomputed $\bar{b} = \lfloor 2^{15} b/q \rceil$.[2] Under this choice, Equation 2.1 guarantees $|ab - eq| \leq q$ for any 16-bit signed integer a. (To see this, note that $|a| \leq 2^{15}$, $\left|\mathrm{mod}^{\lfloor \cdot \rceil} q\right| = (q-1)/2 < q/2$ and $\left|\mathrm{mod}^{\lfloor \cdot \rceil} R\right| = 1/2$.) This allows us to compute $ab - eq$ using only 16-bit arithmetics, specifically one `mul` for the lower-half product $[ab]_l = ab \bmod 2^{16}$, one `sqrdmulh` for e, and lastly one fused-multiply-subtract `mls` for $[[ab]_l - eq]_l = [ab - eq]_l = ab - eq$. Note that we are effectively computing in $\mathbb{Z}_{2^{16}}$, so *overflows are destined but completely irrelevant*.

2.3 CRT and FFTs/NTTs

The CRT (Chinese Remainder Theorem) is a standard theorem of ring theory that relates quotient rings for coprime ideals $I_1, I_2 \subset R$ and their product[3] $I = I_1 I_2$. Specifically, it states that R/I is isomorphic to the product of rings $(R/I_1) \times (R/I_2)$. (Recall that I_1, I_2 being coprime ideals means that $f + g = 1$ for some $f \in I_1$ and $g \in I_2$, and that the product $I_1 I_2$ is the ideal generated by ring elements of the form fg, or more verbosely

[1] They showed only the case where $[\![\cdot]\!]_1 = \lfloor \cdot \rceil$, and stated their bound [6, Cor. 2] without the notion of $\left|\mathrm{mod}^{[\![\cdot]\!]} N\right|$. This notion and the generalized bound are first introduced in [18].

[2] In [6] they chose $R = 2^{16}$, $[\![\cdot]\!]_0 = 2\lfloor \cdot/2 \rceil$ ("round to even") and $[\![\cdot]\!]_1 = \lfloor \cdot \rceil$, with precomputed $\hat{b} = [\![2^{16} b/q]\!]_0/2$. Despite the different appearance, both choices lead to exactly the same computation.

[3] We are assuming the ring's commutativity. For non-commutative rings, one has to instead define I as the intersection $I_1 \cap I_2$.

doi:10.6342/NTU202501941

$I_1 I_2 = \{\sum_{i=1}^{n} f_i g_i \mid f_i \in I_1, g_i \in I_2 \text{ for all } i\}$.)

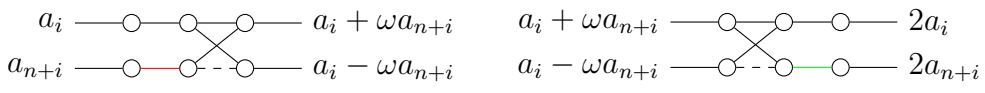
The standard Cooley-Tukey FFT (Fast Fourier Transform) or NTT (Number Theoretic Transform) can be regarded as a special class of instances of CRT. Specifically, it follows from CRT that the mapping

$$\mathbb{Z}_q[x]/\langle x^{2n} - \omega^2 \rangle \to \mathbb{Z}_q[x]/\langle x^n - \omega \rangle \times \mathbb{Z}_q[x]/\langle x^n + \omega \rangle$$
$$f \mapsto (f \bmod (x^n - \omega), f \bmod (x^n + \omega))$$

is an isomorphism. Indeed, by rewriting in terms of the coefficients

$$(a_0, \ldots, a_{n-1}, a_n, \ldots, a_{2n-1}) \mapsto \big((a_0 + \omega a_n, a_1 + \omega_{n+1}, \ldots, a_{n-1} + \omega a_{2n-1}),$$
$$(a_0 - \omega a_n, a_1 - \omega_{n+1}, \ldots, a_{n-1} - \omega a_{2n-1})\big),$$

we see that this mapping is easily invertible. As shown in Figure 2.1, the Cooley-Tukey butterfly computes $(a_0 + \omega a_n, a_0 - \omega a_n)$ from (a_0, a_n), while the Gentleman-Sande butterfly computes the inverse (up to a factor of 2).



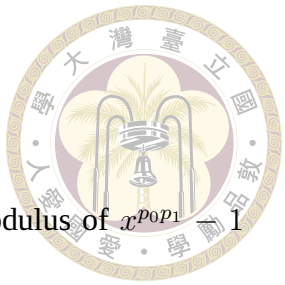
(a) Cooley–Tukey (CT) butterfly (b) Gentleman–Sande (GS) butterfly

Figure 2.1: Butterflies in radix-2 NTT; the red and green edge represent multiplication by ω and ω^{-1} respectively

If we repeat the above process enough times, we get the NTTs used for ML-KEM/DSA. In short CT and GS butterflies suffices for ML-KEM and ML-DSA, but for NTRU Prime, maximal speed requires some other decompositions, which we shall describe below.

doi:10.6342/NTU202501941

2.3.1 Good–Thomas



Good and Thomas [14, 24] proposed turning an NTT for a modulus of $x^{p_0 p_1} - 1$ with coprime p_0, p_1 into a two-dimensional NTT by substituting $x = yz$ with $y^{p_0} = z^{p_1} = 1$. More precisely, the univariate polynomial $\sum_{i=0}^{p_0 p_1} a_i x^i$ becomes a bivariate $\sum_{j=0}^{p_0-1} \sum_{k=0}^{p_1-1} a_{R(j,k)} y^j z^k$, where $R(j,k) = \left(jp_1 \left(p_1^{-1} \bmod p_0\right) + kp_0 \left(p_0^{-1} \bmod p_1\right)\right) \bmod p_0 p_1$ is the "Ruritanian permutation". With the "Good's Trick", we can do the p_0-NTT (in the y-axis) and a the p_1-NTT (in the z-axis) independently. In particular, both these FFTs are in a ring modulo $y^{p_0} - 1$ and $z^{p_1} - 1$, which makes things a lot simpler and more repetitive (i.e., with fewer constants to load).

2.3.2 Rader

Rader's FFT [22] offers a way to compute p-FFT for odd primes p by reducing the problem into a size-$(p-1)$ convolution. We can then transform the problem back to FFTs, where their size $(p-1)$ will at least have 2 as a factor. The basic p-ary transform procedures, usually also called *butterflies*, can be derived from the Rader procedure. For small p, it is worthwhile to optimize the resulting butterfly by hand using some equalities and we do so using various cyclotomic equalities such as $\sum_{i=1}^{p-1} \omega_p^i = -1$. In our algorithm, we use radix-3 and -5 butterflies as basic building blocks. We write $F_i = \sum_{j=0}^{p-1} f_j \omega_p^{ij}$, $\hat{F}_i = F_i - f_0$.

Radix-3 To avoid clutter, we abbreviate ω_3 to ω during our discussion of the radix-3 case. Here, Rader's trick gives the equation $(\hat{F}_1, \hat{F}_2) = (f_1, f_2) * (\omega, \omega^2)$. Using FFT to

doi:10.6342/NTU202501941

compute the convolution, we get

$$a_0 = \hat{F}_1 + \hat{F}_2 = (f_1 + f_2)(\omega + \omega^2) = -(f_1 + f_2) \tag{2.2a}$$

$$a_1 = \hat{F}_1 - \hat{F}_2 = (f_1 - f_2)(\omega - \omega^2) \tag{2.2b}$$

$$2\hat{F}_1 = a_0 + a_1 \tag{2.2c}$$

$$2\hat{F}_2 = a_0 - a_1\,. \tag{2.2d}$$

There is no need to remove the factor of 2 here. We can absorb this factor into all of the outputs F_0, F_1, F_2. With this in mind, we have

$$2F_0 = 2(f_0 + f_1 + f_2) \tag{2.2e}$$

$$2F_1 = 2f_0 + 2\hat{F}_1 \tag{2.2f}$$

$$2F_2 = 2f_0 + 2\hat{F}_2\,. \tag{2.2g}$$

Note that we have the intermediate value $f_1 + f_2$ during (2.2a), which can be reused during (2.2e). Also, rather than adding $2f_0$ into $2\hat{F}_1$ and $2\hat{F}_2$ at (2.2f) and (2.2g), we can save an addition by letting $a_0 \leftarrow a_0 + 2f_0$ before computing (2.2c) and (2.2d). The resulting algorithm can be drawn as a butterfly shown in Figure 2.2.

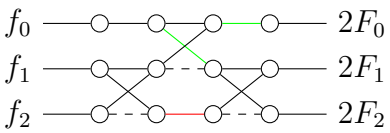


Figure 2.2: Our radix-3 butterfly; the green and red edges represent multiplication by 2 and $\omega_3 - \omega_3^2$ respectively

Radix-5 During our discussion of the radix-5 case, we abbreviate ω_5 to ω. This time, Rader's trick gives the equation $(\hat{F}_1, \hat{F}_3, \hat{F}_4, \hat{F}_2) = (f_1, f_2, f_4, f_3) * (\omega, \omega^3, \omega^4, \omega^2)$. Ap-

doi:10.6342/NTU202501941

plying 2-NTT once gives

$$(a_0, a_1) = (\hat{F}_1 + \hat{F}_4, \hat{F}_3 + \hat{F}_2) = (f_1 + f_4, f_2 + f_3) * (\omega + \omega^4, \omega^3 + \omega^2)$$
$$(a_2, a_3) = (\hat{F}_1 - \hat{F}_4, \hat{F}_3 - \hat{F}_2) = (f_1 - f_4, f_2 - f_3) \,\overline{*}\, (\omega - \omega^4, \omega^3 - \omega^2),$$

where the operator $\overline{*}$ denotes negacyclic convolution or, equivalently, multiplication modulo $x^2 + 1$ when (u, v) is identified with polynomial $u + vx$. Note that while we are able to decompose the cyclic convolution again:

$$b_0 = a_0 + a_1 = (f_1 + f_4 + f_2 + f_3)(\omega + \omega^4 + \omega^3 + \omega^2) = -(f_1 + f_4 + f_2 + f_3)$$
$$b_1 = a_0 - a_1 = (f_1 + f_4 - f_2 - f_3)(\omega + \omega^4 - \omega^3 - \omega^2)$$
$$2a_0 = b_0 + b_1$$
$$2a_1 = b_0 - b_1\,,$$

the absence of square root of -1 inside $\mathbb{Z}_q = \mathbb{Z}_{4591}$ forbids us to do the same to the negacyclic convolution. Nonetheless, by falling back to Karatsuba (and rescaling everything here by 2 to meet the factor in the cyclic part), we obtain

$$2c_0 = (f_1 - f_4) \cdot 2(\omega - \omega^4)$$
$$2c_\infty = (f_2 - f_2) \cdot 2(\omega^3 - \omega^2)$$
$$2c_1 = (f_1 - f_4 + f_2 - f_3) \cdot 2(\omega - \omega^4 + \omega^3 - \omega^2)$$
$$2a_2 = 2c_0 - 2c_\infty$$
$$2a_3 = 2c_1 - 2c_0 - 2c_\infty\,.$$

doi:10.6342/NTU202501941

Combined together, these grant us

$$
\begin{aligned}
4F_0 &= 4(f_0 + f_1 + f_2 + f_3 + f_4) \\
4F_1 &= 4f_0 + 4\hat{F}_1 = 4f_0 + 2a_0 + 2a_2 \\
4F_4 &= 4f_0 + 4\hat{F}_4 = 4f_0 + 2a_0 - 2a_2 \\
4F_3 &= 4f_0 + 4\hat{F}_3 = 4f_0 + 2a_1 + 2a_3 \\
4F_2 &= 4f_0 + 4\hat{F}_2 = 4f_0 + 2a_1 - 2a_3\,.
\end{aligned}
$$

Corresponding to the radix-3 case, by eliminating the common subexpression $f_1 + f_2 + f_3 + f_4$, and replacing the four additions with $4f_0$ with one in $b_0 \leftarrow b_0 + 4f_0$, we end up with a radix-5 butterfly shown in Figure 2.3, using 4 modular multiplications in total. Though be aware that it is impossible to do every multiply-by-4 edge by simply left-shift by 2: One can easily see that overflows are destined to occur in some of the nodes, even if $f_0, \ldots, f_4$ are all in the tightest range $[-2295, 2295]$. One solution to this is treat this as a Barrett multiplication by 4, except with the actual multiply by 4 replaced with a shift left 2.

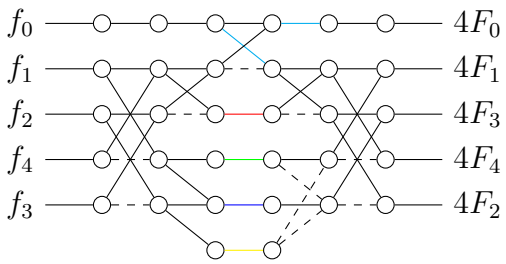


Figure 2.3: Our radix-5 butterfly; the cyan, red, green, blue and yellow edges represent multiplication by 4, $\omega_5 + \omega_5^4 - \omega_5^3 - \omega_5^2$, $2(\omega_5 - \omega_5^4)$, $2(\omega_5 - \omega_5^4 + \omega_5^3 - \omega_5^2)$ and $2(\omega_5^3 - \omega_5^2)$ respectively

doi:10.6342/NTU202501941

2.4 Doing iNTT as NTT

Recall that the n-NTT is defined as $F_j = \sum_{i=0}^{n-1} \omega_n^{ij} f_i$, where ω_n is a principal n-th root of unity. Its inverse n-iNTT can be computed as $f_i = \frac{1}{n} \sum_{j=0}^{n-1} \omega_n^{-ij} F_j$. Barring the factor of $1/n$, the two formulae are strikingly symmetrical. Making use of this symmetry, there are several common ways to adapt algorithm for NTT into one for iNTT (up to a constant factor).

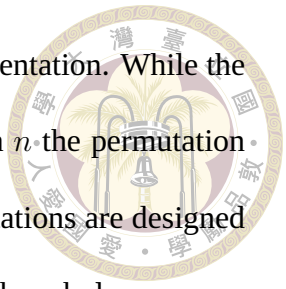
Consider the naïve algorithm for NTT. By replacing ω_n^k with ω_n^{-k} in each multiply-by-constant step, one obtains an algorithm for iNTT. Frequently, the coefficients ω_n^k (along with the precomputed constants that Barrett multiplication requires) are organized into a look-up table. In this case, the substitution amounts to swapping the look-up table. We remark that similar "look-up table swapping" tricks also exist for typical FFT algorithms (including Cooley-Tukey, Good-Thomas and Rader), since their correctness only depend on cyclotomic equalities (i.e. $\sum_{i=0}^{d-1} \omega_n^{in/d} = 0$ for $d \mid n$) which are invariant under the mapping $\omega_n^k \mapsto \omega_n^{-k}$.

Alternatively, we can "reflect the circular sequence F_j with respect to F_0" before feeding it to the NTT algorithm as the input sequence. This works because

$$\begin{aligned}\sum_{i=0}^{n-1} \omega_n^{ij} F_{(n-i) \bmod n} &= F_0 + \sum_{i=1}^{n-1} \omega_n^{ij} F_{n-i} \\ &= F_0 + \sum_{i=1}^{n-1} \omega_n^{(n-i)j} F_i \\ &= \sum_{i=0}^{n-1} \omega_n^{-ij} F_i = n f_j \,,\end{aligned}$$

a fact commonly known as the "duality property of DFT". This trick works for any NTT

doi:10.6342/NTU202501941



algorithm, even when the coefficients are hard-coded into the implementation. While the rearrangement of F_j can be costly when n is large, for small enough n the permutation is essentially free. For example, all of our 10- and 9-iNTT implementations are designed with this method (see Section 3.4.1 for more details). At this size, the whole sequence easily fits inside vector registers without spilling to the stack. As such, we expected the compiler to perform the permutation by simply modifying the destination register of the load instructions appropriately. (We actually went out of our way to inspect the emitted code and to confirm that this is indeed the case.)

2.5 Weighted Convolutions and Toeplitz Matrix-Vector Products

At the end of our reductions we typically come down to a product $h = fg \bmod (x^{16} - w)$ for degree-15 (or lower) polynomials f, g. When $w = 1$ this is the standard convolution and $w = -1$ it is a negacyclic convolution. When $w \neq \pm 1$, we can write this as

$$\begin{bmatrix} h_0 \\ h_1 \\ h_2 \\ \vdots \\ h_{15} \end{bmatrix} = \begin{bmatrix} g_0 & wg_{15} & wg_{14} & \cdots & wg_1 \\ g_1 & g_0 & wg_{15} & \cdots & wg_2 \\ g_2 & g_1 & g_0 & \cdots & wg_3 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ g_{15} & g_{14} & g_{13} & \cdots & g_0 \end{bmatrix} \begin{bmatrix} f_0 \\ f_1 \\ f_2 \\ \vdots \\ f_{15} \end{bmatrix}.$$

In other words, it is a Toeplitz Matrix to Vector Product (TMVP). When w has a square root we can use a Cooley-Tukey NTT to reduce it to two smaller TMVPs. Otherwise, as

doi:10.6342/NTU202501941

in Karatsuba, we need three 8×8 TMVPs using the methods in (e.g.) [10]:

$$\begin{bmatrix} C_0 \\ C_1 \end{bmatrix} = \begin{bmatrix} A_0 & A_{-1} \\ A_1 & A_0 \end{bmatrix} \begin{bmatrix} B_0 \\ B_1 \end{bmatrix} \rightarrow \begin{cases} C_0 &= (A_{-1} - A_0)B_1 + A_0(B_0 + B_1) \\ C_1 &= A_0(B_0 + B_1) + (A_1 - A_0)B_0. \end{cases}$$

Note that submatrices of a Toeplitz matrix are also Toeplitz.

doi:10.6342/NTU202501941

doi:10.6342/NTU202501941

Chapter 3 Implementation

In the following, we denote the two input polynomial as $A(x) = \sum_{i=0}^{760} a_i x^i$ and $B(x) = \sum_{i=0}^{760} b_i x^i$. Our algorithm is for big-by-big multiplications, that is, we can only assume that $a_i, b_i \in [-2295, 2295]$ for each i. Similarly, the output polynomial, before and after reducing modulo $x^{761} - x - 1$, are respectively denoted as $C^{\text{full}}(x) = \sum_{i=0}^{1520} c_i^{\text{full}} x^i$ and $C(x) = \sum_{i=0}^{760} c_i x^i$.

3.1 Choice of Ring

The Old: The NTRU Prime polynomial modulus $x^{761} - x - 1$ forbids the use of straightforward NTT-based multiplications. Nonetheless, NTT-based multiplications are still applicable if we first lift the computation to $\mathbb{Z}_q[x]$, then map to $\mathbb{Z}_q[x]/\langle Q(x)\rangle$ with $\deg(Q) \geq 1521$ for some new polynomial modulus $Q(x)$ that is more NTT-friendly [1, 7, 11]. FFT-based multiplications limit the choice of $Q(x)$, especially when we keep the integral modulus at $q = 4591$. For instance, to do an incomplete NTT on $\mathbb{Z}_q[x,t]/\langle x^k - t, t^n - 1\rangle \cong \mathbb{Z}_q[x]/\langle x^{nk} - 1\rangle$ requires existence of a n-th primitive root of unity in $\mathbb{Z}_q$, hence $n|(4591 - 1) = 2 \cdot 3^3 \cdot 5 \cdot 17$. Suppose we picked $k = 16$ so as to store each base-case polynomial conveniently in two Neon registers. Since $\deg(Q) = nk \geq 1521$, we have $n \geq 96$. To make n as small as possible, it can only reasonably be $102 = 2 \cdot 3 \cdot 17$. This

doi:10.6342/NTU202501941

leads to computation in the ring $\mathbb{Z}_q[x]/\langle x^{1632}-1\rangle$, as in [19]. Later it was improved to use the smaller ring $\mathbb{Z}_q[x]/\langle \Phi_{17}(x^{96})\rangle$ with truncated Rader's FFT ([17]). Alternatively, one could use as in [9] a truncated Schönhage/Nussbaumer NTT to multiply in $\mathbb{Z}_q[x]/\langle (x^{1024}+1)(x^{512}-1)\rangle$. Effectively, their algorithm multiplies modulo $x^{1024}+1$ and modulo $x^{512}-1$ (both subtasks via Schönhage/Nussbaumer) followed by recovering the product in $\mathbb{Z}_q[x]/\langle (x^{1024}+1)(x^{512}-1)\rangle$ via CRT.

The New: We propose a similar approach, except that we use a polynomial modulus of the form $Q(x) = (x^{n_{\text{main}}}-1)x^{n_{\text{low}}}$ for some $n_{\text{main}} > n_{\text{low}}$. In our case, we use $n_{\text{main}} = 1440$ and $n_{\text{low}} = 81$. We refer to the computation modulo $Q^{\text{main}}(x) = x^{n_{\text{main}}}-1$ and $Q^{\text{low}}(x) = x^{n_{\text{low}}}$ as the main and the low part respectively, and denote the product in the two parts as $C^{\text{main}}(x) = \sum_{i=0}^{1439} c_i^{\text{main}} x^i$ and $C^{\text{low}}(x) = \sum_{i=0}^{80} c_i^{\text{low}} x^i$.

Using a power of x as a factor has some advantages. CRT is guaranteed to be applicable since $\gcd(x^{n_{\text{main}}}-1, x^{n_{\text{low}}}) = 1$. In fact, the inverse map will be somewhat simpler than usual. Because

$$\begin{aligned} x^{n_{\text{main}}} &\equiv 1 \pmod{x^{n_{\text{main}}}-1} & 1-x^{n_{\text{main}}} &\equiv 0 \pmod{x^{n_{\text{main}}}-1} \\ x^{n_{\text{main}}} &\equiv 0 \pmod{x^{n_{\text{low}}}} & 1-x^{n_{\text{main}}} &\equiv 1 \pmod{x^{n_{\text{low}}}}, \end{aligned}$$

we have $C^{\text{full}} \equiv x^{n_{\text{main}}} C^{\text{main}} + (1-x^{n_{\text{main}}}) C^{\text{low}} \pmod{Q(x)}$, and

$$c_i^{\text{full}} = \begin{cases} c_i^{\text{low}} & 0 \le i < 81 \\ c_i^{\text{main}} & 81 \le i < 1440 \\ c_{i-1440}^{\text{main}} - c_{i-1440}^{\text{low}} & 1440 \le i < 1521, \end{cases} \tag{3.1}$$

doi:10.6342/NTU202501941

in terms of the coefficients. Moreover, we can control $\deg(Q)$ as needed by adjusting n_{low}. Finally, multiplication in the low part has some special optimization opportunities, since terms with degree at least n_{low} can be safely discarded. In this case, one can can arrive at specialized forms of Karatsuba, Toom-Cook, or inverse NTT in an NTT-based to skip some computations.

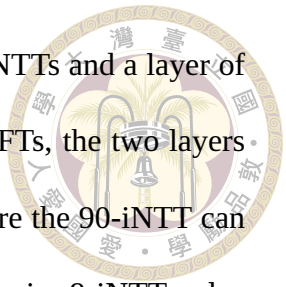
3.2 Algorithmic and Computational Overview

In this overview, we outline the end-to-end structure of the whole algorithm. This begins with multiplication in $\mathbb{Z}_q[x]/\langle(x^{1440}-1)x^{81}\rangle$, which is in turn decomposed into main and low part. To combine the two products into one polynomial of length 1521, we present a way to apply Equation 3.1 while avoiding excessive data movements. After reducing modulo $x^{761}-x-1$, the final step is to "freeze" the coefficients into the canonical representation range $[-2295, 2295]$. Each of the stages are implemented with one or more subroutines, most of which have a corresponding C++ function.

3.2.1 Transformation Process of Main Part

For the main part, we use an incomplete NTT with $n = 90, k = 16$. In other words, we start from the isomorphic ring $\mathbb{Z}_q[x,t]/\langle x^{16}-t, t^{90}-1\rangle \cong \mathbb{Z}_q[x]/\langle x^{1440}-1\rangle$, then use a 90-NTT to map to $\prod_{i=0}^{89}\mathbb{Z}_q[x]/\langle x^{16}-\omega_{90}^i\rangle$. The 90-NTT is in turn decomposed into 10-NTTs and 9-NTTs using Good-Thomas NTT. More concretely, for the 90 inputs $f_0, f_1, \ldots, f_{89}$ to the 90-NTT, by rearranging them into a 2D array of shape 10×9 as Figure 3.1 (a), the task is reduced to a 2D NTT. The initial ("Good's") and final ("Ruritanian") permutations are always merged with the computations of another layer.

doi:10.6342/NTU202501941



The 2D NTT can be computed with a layer of column-wise 10-NTTs and a layer of row-wise 9-NTTs. Moreover, by separability of multidimensional DFTs, the two layers can be done in either order. A similar situation exists in reverse, where the 90-iNTT can be implemented as a combination of column-wise 10-iNTTs and row-wise 9-iNTTs, also in either order. Here, we chose to do column-wise 10-NTT and row-wise 9-iNTTs first in the forward and inverse transform respectively. Why so will be explained in Section 3.3.1.

Note to those reading code: in subroutine names, "backward" (abbr. `bwd`) means iNTT (inverse NTT), and "forward" describe the forward transform of the NTT (abbr. `fwd`); Layer 1 and 2 respectively represent a column-wise 10-(i)NTT and a row-wise 9-(i)NTT. So the main part transformations (i.e. excluding the base-case convolutions) consist of 4 subroutines, each named `main_lay{1,2}__{f,b}wd` as appropriate.

3.2.2 Transformation Process of Low Part

Note that a_i, b_i have no effect on the output if $i \geq 81$. Moreover, a_{80}, b_{80} only affect the output with the terms $a_{80}b_0x^{80}$ and $a_0b_{80}x^{80}$, which we compute separately and otherwise ignore the degree-80 term of the inputs for now. This leaves us with the two degree-79 input polynomials $\sum_{i=0}^{79} a_ix^i$ and $\sum_{i=0}^{79} b_ix^i$. Since the degree of their product is now at most 158, we compute it in $\mathbb{Z}_q[x]/\langle x^{160} - 1\rangle$. With a 10-NTT, this is mapped to $\prod_{i=0}^{9} \mathbb{Z}_q[x]/\langle x^{16} - \omega_{10}^i\rangle$. After base-case convolutions and 10-iNTT, we obtain a polynomial of length 160. Discarding the terms with degrees at least 81 gives us $(\sum_{i=0}^{79} a_ix^i)(\sum_{i=0}^{79} b_ix^i) \bmod x^{81}$. Finally adding back the ignored contribution of (a_{80}, b_{80}), we get our desired low part product

$$C^{\text{low}} = AB \bmod x^{81} = \left(\sum_{i=0}^{79} a_ix^i\right)\left(\sum_{i=0}^{79} b_ix^i\right) \bmod x^{81} + (a_{80}b_0 + a_0b_{80})x^{80}. \quad (3.2)$$

doi:10.6342/NTU202501941

The 10-(i)NTT are implemented as subroutines `low_lay1__{f,b}wd`.

3.2.3 Base-case Weighted Convolutions

At the end of the forward NTT, the main and the low part are decomposed into 90 and 10 base-cases respectively. These are all in the form of size-16 weighted convolutions [12], that is, multiplication between two length 16 polynomials, modulo some polynomial of the form $x^{16} - \omega_{90}^{i}$ for some i. (Note that ω_{10}^{i} can be rewritten as ω_{90}^{9i}.) The coefficient ω_{90}^{i} will be referred as the weight of the weighted convolutions. Despite their similarity at first glance, it is better to implement the base-cases of each part as separate subroutines. Indeed, the NTTs located before the base-cases are different in the two parts, and so are the iNTTs located after. Implementing the base-cases separately allows us to plan the modular reductions differently, adaptive to the different input guarantees/output requirements on the coefficient ranges. Additionally, there are different extra operations that we need to perform in this stage. These are some skipped pre/postprocessing steps from the surrounding (i)NTTs, strategically rescheduled and absorbed into computations here. The details and benefits of this rearrangement will be presented in Section 3.3. In summary, the base-case weighted convolutions are implemented as 2 independent subroutines, named `basemul__{main,low}`.

3.2.4 CRT with Minimal Data Movements

As we discussed in Section 3.1, our choice of the two coprime polynomial moduli $x^{1440} - 1$ and x^{81} admits a remarkably simple inverse CRT map. In fact, by rewriting Equation 3.1 into imperative statements as in Algorithm 1, we see that to combined the prod-

doi:10.6342/NTU202501941

ucts from the two parts, it suffices to (a) store `C_main` into `C_full[81 : 1521]` (in a segmented fashion), (b) store `C_low` into `C_full[0 : 81]` and finally (c) subtract `C_low` from `C_full[1440 : 1521]`. Note that apart from (c), the steps amounts to exactly 1521 element-wise storages.

Algorithm 1 Inverse CRT map

Require: Polynomials $C^{\text{main}}, C^{\text{low}}$ stored as `C_main[1440]`, `C_low[81]`
Ensure: Polynomial C^{full} stored as `C_full[1521]`

```
1: C_full[81 : 1440] = C_main[81 : 1440]
2: C_full[1440 : 1521] = C_main[0 : 81]
3: C_full[0 : 81] = C_low[0 : 81]
4: C_full[1440 : 1521] -= C_low[0 : 81]
```

Given the simplicity of this process, it is conceivable to merge the steps into the last layers of the inverse transform, i.e. `{main,low}_lay1__bwd`. More specifically, after the computation of 10-iNTT, right when we are about to insert the coefficients into `C_main` and `C_low`, we instead directly store the coefficients into `C_full`. For the remaining step (c), we merge it into `low_lay1__bwd` as well. In the end, compared to implementing the CRT formula as a separate subroutine, we effectively saved $2 \cdot 1521$ element-wise load and store instructions, or at least $2 \cdot \lceil 1521/32 \rceil = 96$ vectorized instructions `ld1` and `st1` (which can simultaneously operate on 4 vector registers at most), just to give an idea.

We should point out that this is still a slight simplification from what we really implemented in our code: Because of the nature of Neon instructions, where 8 coefficients must be manipulated as a whole, and some further complications like application of Equation 3.2, there are many special-cases/overlaps/leftovers that need to treated carefully, frequently with dedicated scalar instructions. Nevertheless, this technique still saves considerable amount of data movements, and its correctness is verified in CryptoLine just the same.

doi:10.6342/NTU202501941

3.2.5 Final Reductions and Freezing Coefficients

After the iNTT we need to do three things to compute the final result: multiply by a constant, reduce by modulo $x^{761} - x - 1$, reduce modulo $q = 4591$ to the canonical representative (an operation termed "freezing" as opposed to "squeezing", or lazy reduction, in [8]). Clearly one should reduce $\sum_{i=0}^{1520} c_i^{\text{full}} x^i$ by $x^{761} - x - 1$ first which gets us $(c_{1520}^{\text{full}} + c_{760}^{\text{full}})x^{760} + (c_{1520}^{\text{full}} + c_{1519}^{\text{full}} + c_{759}^{\text{full}})x^{759} + (c_{1519}^{\text{full}} + c_{1518}^{\text{full}} + c_{758}^{\text{full}})x^{758} + \cdots + (c_{762}^{\text{full}} + c_{761}^{\text{full}} + c_1^{\text{full}})x + (c_{761}^{\text{full}} + c_0^{\text{full}})$, then do a Barrett multiply by the constant (in our code, $1/170$) followed by a final conditional addition to reduce to between ± 2295.

3.3 Optimization Opportunities

With the general algorithm structure in mind, now we give a deeper dive into the some of the implementation details. In particular, we will focus on the various optimization opportunities noticeable in the computation steps, and describe our ways to exploit these observations to save time.

3.3.1 Zero-skipping

During the forward transform of the main part, the 90 inputs $f_0, f_1, \ldots, f_{89}$ to the 90-NTT come from the zero-padded polynomial of length 1440. Specifically, every 16 consecutive coefficients are clumped together, which form the 90 polynomials of length 16 each. Since the polynomial before the zero-padding was only of length 761, we see that only the first $\lceil 761/16 \rceil = 48$ of them $f_0, f_1, \ldots, f_{47}$ are nonzero. By exploiting the zero elements which constitute almost half of the inputs, it is possible to greatly simplify

doi:10.6342/NTU202501941

the forward transform. Now, recall that the 90-NTT was implemented as two layers, and that column-wise 10-NTT and row-wise 9-NTT layer in theory could be done in either order. However, observe that only the preceding layer can make use of the zero elements. Indeed, after its "intermixing" of the coefficients in each column/row, the 2D array will (in general) no longer contain any zeros for the following layer to exploit. Even though the total number of zero elements is the same, depending on the order of the layers, there tends to be a big difference on the saving of computations. After careful comparisons, we concluded that the column-wise 10-NTT makes better use of the zero-inputs, thus should be placed as the frontmost layer. As we will show in Section 3.4.1, zero-inputs in 10-NTTs, especially when they are in specific patterns, will let us skip some modular multiplications.

f_0	f_{10}	f_{20}	f_{30}	f_{40}	f_{50}	f_{60}	f_{70}	f_{80}
f_{81}	f_1	f_{11}	f_{21}	f_{31}	f_{41}	f_{51}	f_{61}	f_{71}
f_{72}	f_{82}	f_2	f_{12}	f_{22}	f_{32}	f_{42}	f_{52}	f_{62}
f_{63}	f_{73}	f_{83}	f_3	f_{13}	f_{23}	f_{33}	f_{43}	f_{53}
f_{54}	f_{64}	f_{74}	f_{84}	f_4	f_{14}	f_{24}	f_{34}	f_{44}
f_{45}	f_{55}	f_{65}	f_{75}	f_{85}	f_5	f_{15}	f_{25}	f_{35}
f_{36}	f_{46}	f_{56}	f_{66}	f_{76}	f_{86}	f_6	f_{16}	f_{26}
f_{27}	f_{37}	f_{47}	f_{57}	f_{67}	f_{77}	f_{87}	f_7	f_{17}
f_{18}	f_{28}	f_{38}	f_{48}	f_{58}	f_{68}	f_{78}	f_{88}	f_8
f_9	f_{19}	f_{29}	f_{39}	f_{49}	f_{59}	f_{69}	f_{79}	f_{89}

(a) All 90 inputs rearranged

f_0	f_{10}	f_{20}	f_{30}	f_{40}				
	f_1	f_{11}	f_{21}	f_{31}	f_{41}			
		f_2	f_{12}	f_{22}	f_{32}	f_{42}		
			f_3	f_{13}	f_{23}	f_{33}	f_{43}	
				f_4	f_{14}	f_{24}	f_{34}	f_{44}
f_{45}					f_5	f_{15}	f_{25}	f_{35}
f_{36}	f_{46}					f_6	f_{16}	f_{26}
f_{27}	f_{37}	f_{47}					f_7	f_{17}
f_{18}	f_{28}	f_{38}						f_8
f_9	f_{19}	f_{29}	f_{39}					

(b) First 48 inputs rearranged

Figure 3.1: 90-NTT, reduced to a (10×9)-NTT with Good-Thomas; the empty locations in (b) shows the distribution of zero-elements

In `main_lay1__fwd` there are of 9 instances of 10-NTT, each operating on a column of the 10×9 array. As explained in the last paragraph, we know beforehand that 42 elements in the array are actually zeros. From Figure 3.1 (b), we see that each 10-NTT instance will contain 4 or 5 zero-inputs depending on the column-index. Specifically, the first 3 columns have 4 zero-inputs each, and the remaining 6 columns have 5 each. However, since each column's zero-inputs are located differently, one would have to implement

doi:10.6342/NTU202501941

each 10-NTT instance separately in order to fully utilize all of the zero-inputs. In certain embedded environments, the increased code size is prohibitive. This is a drawback of Good's NTT. For example, in [1, Section 4.1], the authors used 4 designs of 3-level radix-2 NTT. As the sequence of the required design for each instance are highly irregular, a code generating script was used within their Good's NTT implementation.

To circumvent this, we proposed to cyclically shift the coefficients within each column, then compute the 10-NTT just as usual. Provided we apply a postprocessing step according to the "Time Shifting" property of DFT, we will nevertheless end up with the correct output. More precisely, the property states that if two sequences of length n

$$f[0], f[1], \ldots, f[n-1] \leftrightarrow F[0], F[1], \ldots, F[n-1]$$

form an input/output pair of DFT, then left-shifting f by m results in the pair

$$\begin{aligned} &f[m], f[m+1], \ldots, f[n-1], f[0], \ldots, f[m-1] \\ &\leftrightarrow F[0], \omega_n^{-m} F[1], \omega_n^{-2m} F[2], \ldots, \omega_n^{m} F[n-1]\,. \end{aligned}$$

In NTT parlance, the process of element-wise multiplication of F with $1, \omega_n^{-m}, \omega_n^{-2m}, \ldots, \omega_n^{m}$ is "twisting" the sequence, and the coefficients will be referred as the twisting factors. Hence an intuitive interpretation is that "shifted inputs lead to twisted outputs".

With this technique, we only need two specialized 10-NTT designs. Each of them will handle all the columns with 4 or 5 consecutive zeros, respectively. Moreover, we take this chance to strategically shift the consecutive zeros to appropriate locations, such that the 10-NTT is simplified most effectively. Note that the best choice here depends heavily on the original 10-NTT. For our design, detailed later in Section 3.4.1, it is most beneficial

doi:10.6342/NTU202501941

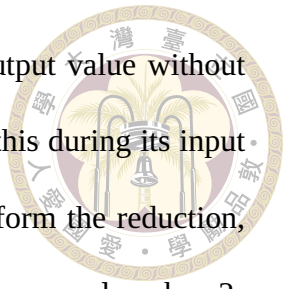
to put the 4 zeros at $f[3], f[4], f[5], f[6]$, and the 5 zeros at $f[3], f[4], f[5], f[6], f[7]$.

We place the postprocessing step at the end of `main_lay1__fwd`. Because of the nonidentical shiftings applied to them, the columns require separate sets of twisting factors each. It is true that this postprocessing step incurs extra modular multiplications. Though we should keep in mind that originally a step of modular reductions would be required around here anyway. As we are replacing it with modular multiplications, the overhead will be comparatively insignificant. Indeed, the merits of simplified NTT and reduced code size more than make up for the postprocessing cost.

In `low_lay1__fwd`, the length 80 input are also padded to length 160, so a similar zero-skipping trick is available. There is only one instance of 10-NTT needed, so we do not have to worry about multiple zero-input patterns for 10-NTT. Nonetheless, we still shift the zeros from $f[5], \ldots, f[9]$ to $f[3], \ldots, f[7]$, which helps to save more modular multiplications, and enables us to reuse the same 10-NTT design in `main_lay1__fwd`. The postprocessing step needed here, i.e. element-wise multiplication of $F[0], \ldots, F[9]$ with the twisting factors $1, \omega_{10}^2, \omega_{10}^4, \ldots, \omega_{10}^{18} = \omega_{10}^8$, is postponed and merged with the base-case convolutions, more specifically the step where we narrow down the result of convolutions from 32-bit to 16-bit.

Compared to `main_lay1__fwd`, as no postprocessing is done within `low_lay1__fwd`, direct outputs of the 10-NTT will have a very wide range of possible values. This makes it impossible to do another round of addition/subtraction without risking overflows in the worst case. It is thus natural to schedule a modular reduction here. However, we argue that in our case, putting this step in the input stage of the base-case subroutine would be the better approach. First, since the weighted convolutions are computed in 32-bit

doi:10.6342/NTU202501941



arithmetic, it is possible that the subroutine can handle our large output value without problems. Moreover, even if reduction is really necessary, handling this during its input stage allows the base-case subroutine to choose the best way to perform the reduction, depending on the polynomial modulus for that specific base-case. For example, when 2-NTT-based algorithm is used for the weighted convolution, which is applicable to only half of the base-cases (See Section 3.3.3), some of the coefficients will, before anything else, be 16-bit modular-multiplied with a constant. This entirely eliminates the need to perform reductions on these specific coefficients.

3.3.2 Early-dropping

In `low_lay1__bwd`, the inverse 10-NTT outputs 10 polynomials $f[0], \ldots, f[9]$, each of length 16. Merging all of them will give us a polynomial of length 160. However, since we are computing modulo x^{81}, only the first 6 polynomials $f[0], \ldots, f[5]$ are needed to recover the lowest 81 coefficients. In this case, we can again simplify the inverse NTT, this time by preemptively dropping the intermediate values which are depended only by the to-be-discarded outputs. To make the most out of this, we use the time shifting property once again, though this time we will instead need to preprocess the inputs, i.e. multiply them element-wise with twisting factors. By doing this, we effectively shift the locations of the discarded outputs. As a concrete example, we ultimately want an inverse 10-NTT that discards $f[6], \ldots, f[9]$. We achieve this by using an inverse 10-NTT that discards $f[3], \ldots, f[6]$, plus a preprocessing step on the inputs where they are multiplied element-wise with $1, \omega_{10}^{7}, \omega_{10}^{14}, \ldots, \omega_{10}^{63} = \omega_{10}^{3}$. Correspondingly, the preprocessing is also "preponed" and merged with the base-case convolutions.

doi:10.6342/NTU202501941

3.3.3 Base-case Convolutions

For the $90 + 10 = 100$ base-case multiplications, these are all in the form of size-16 weighted convolutions [12]. In other words, each of them is a multiplication between two length 16 polynomials, in some ring of the form $\mathbb{Z}_q/\langle x^{16} - \omega_{90}^i \rangle$ for some i. (Note that ω_{10}^i can be rewritten as ω_{90}^{9i}.) As described in Section 2.5, we use 2-NTT or Karatsuba to further reduce the problem into size-8 TMVPs [10]. More specifically, whenever the weight ω_{90}^i has a square root, i.e. $i = 2k$ is even, we are able to apply 2-NTT and reduce the task to multiplication in $\mathbb{Z}_q/\langle x^8 \pm \omega_{90}^k \rangle$, implemented as 2 TMVPs. For the other cases, we have to fallback to Karatsuba, which is less preferable because 3 TMVPs are required.

3.4 Basic Procedures

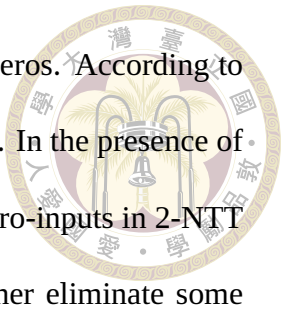
With the aforementioned steps to progressively decompose the problem into smaller subtasks, the remaining pieces of the puzzle are the various basic procedures. As the heart of the whole algorithm, these building blocks are critical to the overall performance of the polynomial multiplier.

3.4.1 NTT designs

10-NTT Present in various subroutines, the 10-NTT designs are all based on reduction to (5×2)-NTT via Good-Thomas. Depending the the specific use case, we adaptively choose the first axis to transform on, in similar essence to Section 3.3.1. With the following examples, we aim to illustrate the utility of this flexibility.

Consider the 10-NTTs in `main_lay1__fwd`. By our manipulation in Section 3.3.1,

doi:10.6342/NTU202501941



we assume that the inputs $f[3]$ to $f[6]$, sometimes even $f[7]$, are all zeros. According to Good-Thomas, these zeros are rearranged as in Figure 3.2 (b) and (c). In the presence of these zeros, it is better to do 5-NTTs first. The reason is that, while zero-inputs in 2-NTT save additions or subtractions, in 5-NTT they occasionally can further eliminate some of the modular multiplications. For example, as shown in Figure 3.3, several additions/ subtractions and one multiplication are saved if both f_1, f_4 are zeros. Similar saving is also achieved when both f_2, f_3 are zeros. In the end, from the locations of the zeros in each columns, we see that at least a multiplication in the first column for both cases, and a multiplication in the second column for the 5 zeros case are saved. This is the ultimate reason for our placement of the 4 and 5 zeros. As remarked in the last 2 paragraphs of Section 3.3.1, we reuse the 5-zeros design again in `low_lay1__fwd` to profit once more from the eliminated modular multiplications.

f_0	f_5
f_6	f_1
f_2	f_7
f_8	f_3
f_4	f_9

(a) Original input layout

f_0	
	f_1
f_2	f_7
f_8	
	f_9

(b) 4 zeros case

f_0	
	f_1
f_2	
f_8	
	f_9

(c) 5 zeros case

Figure 3.2: 10-NTT, reduced to a (5×2)-NTT with Good-Thomas

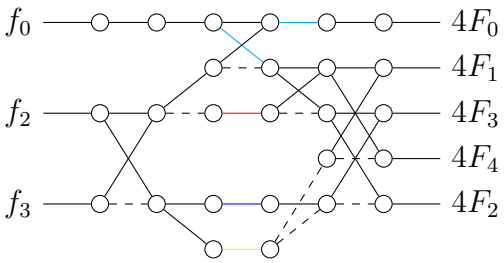
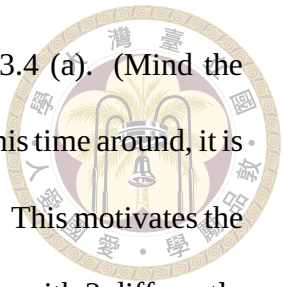


Figure 3.3: Our radix-5 butterfly, pruned in the case that f_1, f_4 are zeros

For the 10-iNTTs in `low_lay1__bwd`, on the other hand, we need an inverse 10-NTT that discards $f[3], \ldots, f[6]$. We achieved this by modifying a forward 10-NTT that discards $F[3], \ldots, F[6]$ (See Section 2.4). Recall that in the further reduction to $(5 \times$

doi:10.6342/NTU202501941



2)-NTT, the output locations is also rearranged as shown in Figure 3.4 (a). (Mind the difference in indexing, cf. Figure 3.2 (a).) More importantly, note that this time around, it is the later transformed axis that can benefit from the discardable outputs. This motivates the design where we perform the 5 instances of 2-NTT first, and follow up with 2 differently pruned instances of 5-NTT. From Figure 3.4 (b) we see that the first and second column needs radix-5 butterfly designs that discards F_2, F_3 and F_0, F_4 respectively. Illustrated in Figure 3.5, the former butterfly skips a modular multiplication originally in the radix-5 butterfly. This additional saving is the reason behind the manipulation described in Section 3.3.2.

It is important to note that, because of the constant factor of 4 in our 5-NTT, the output of our 10-NTT is also scaled up by 4. For the 10-iNTT on the other hand, the "iNTT as NTT" trick further introduced a factor of 10, resulting in the final constant factor of 40.

F_0	F_5
F_2	F_7
F_4	F_9
F_6	F_1
F_8	F_3

(a) Original output layout

F_0	
F_2	F_7
	F_9
	F_1
F_8	

(b) 4 discards case

Figure 3.4: 10-NTT, reduced to a (5×2)-NTT with Good-Thomas.

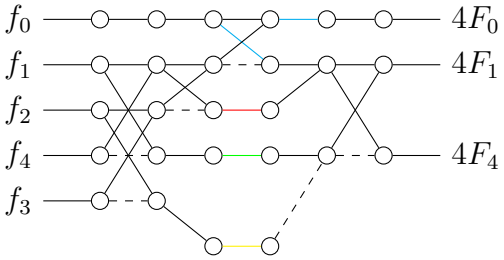
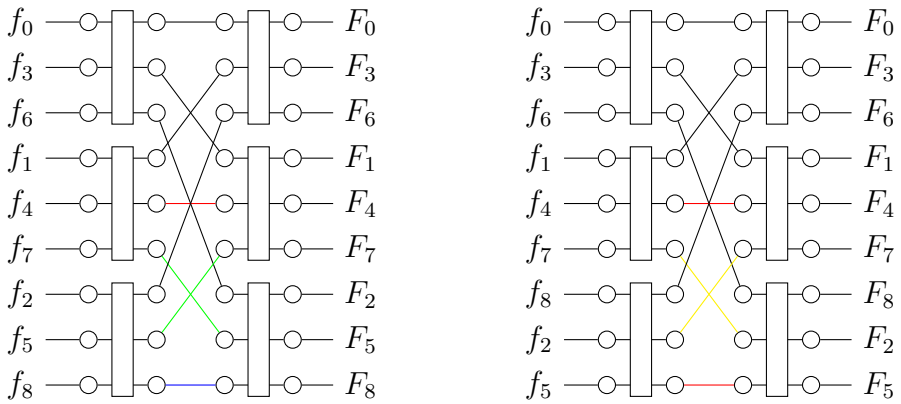


Figure 3.5: Our radix-5 butterfly, pruned in the case that F_2, F_3 can be discarded

9-NTT The most typical way to implement a 9-NTT is to use radix-3 Cooley-Tukey with two levels. To reuse the same 3-NTT on the second level, one needs to twist the

doi:10.6342/NTU202501941

outputs between the levels. The resulting algorithm is shown in Figure 3.6 (a). In our implementation, we use a slightly adjusted variant: By twisting two of the radix-3 "sub-butterflies" as shown in Figure 3.6 (b), only 2 distinct constants are required to perform the inter-level twisting, instead of the original 3. This did not really offer a performance boost, but the freed up register slots are nonetheless beneficial.



(a) Standard radix-3 Cooley-Tukey (b) Our variant

Figure 3.6: 9-NTT designs; the rectangles represent radix-3 "sub-butterflies", and the red, green, blue and yellow edges represent multiplication by ω_9, ω_9^2, ω_9^4 and ω_9^8 respectively

Note that after the two levels of 3-NTT, since our radix-3 butterfly introduce a constant factor of 2, the 9-NTT output is expected to be scaled up by 4. In our implementation though, the factor is 2 instead. This is caused by another slight optimization used in our implementation described below.

Consider the radix-3 butterfly in Figure 2.2. By rescheduling the multiply-by-2 steps, we can "move the green edges around". From the butterfly on the right of Figure 3.7, we can see that if $2f_0$ is readily available, it can absorb a green edge and allow us to implement the butterfly using only one multiply-by-2 step, illustrated in Figure 3.8. Alternatively, as long as f_1, f_2 can be halved conveniently, this alternative butterfly is also applicable. In this case, we are able to eliminate the constant factor of 2 and reduce the output range.

Within our 9-NTT, the alternative radix-3 butterfly was used in the second level. Out

doi:10.6342/NTU202501941

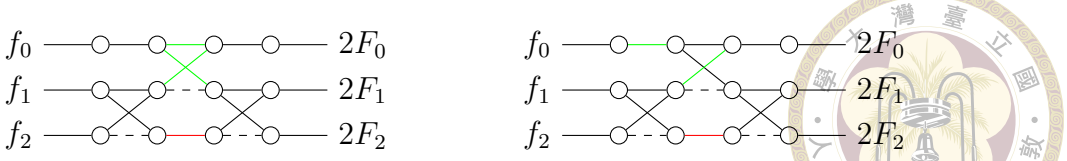


Figure 3.7: Our radix-3 butterfly with the multiply-by-2 steps rescheduled

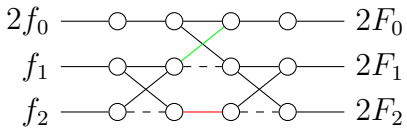


Figure 3.8: Alternative radix-3 butterfly when $2f_0$ is available

of the 9 intermediate values, which is scaled up by 2 during the first level, it suffices to scale 6 of them back down. More specifically, by examining the data flow in Figure 3.6 (b), we see that the 6 intermediate values to scale down are precisely the outputs of the lower 2 sub-butterflies of the first level. This is easily achieved by skipping the multiply-by-2 step for $2F_0$ in the 2 sub-butterflies, and replacing the twisting factors by their halves.

All in all, after expanding every sub-butterflies and appling the above tricks, our 9-NTT implementation is illustrated in Figure 3.9. As noted before, the constant factor is 2, instead of 4 as one might expect. Correspondingly, the 9-iNTT is also has a constant factor of $2 \cdot 9 = 18$.

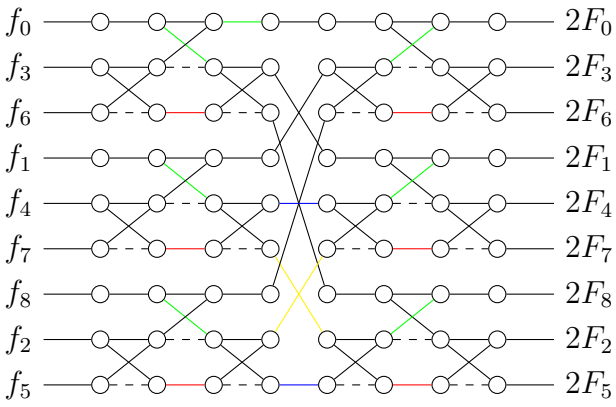


Figure 3.9: Our 9-NTT implementation, shown as a butterfly graph; the red, green, blue and yellow edges represent multiplication by $\omega_3 - \omega_3^2$, 2, $\omega_9/2$ and $\omega_9^8/2$ respectively

doi:10.6342/NTU202501941

3.4.2 Variants of Barrett Reduction/Multiplication

One way to summarize the central idea behind Barrett multiplication is as follows. We begin by approximating ab/q using fixed-point arithmetic. Then the rounded value of the fixed-point approximation will be a reasonable approximation for $\lfloor ab/q \rceil$. For example, the standard Barrett multiplication uses $\bar{b} = \lfloor 2^r b/q \rceil$, which is just the conventional fixed-point representation of b/q with r fraction bits. Continuing this, computing $a\bar{b}$ is just the usual way to compute ab/q while retaining r bits of fractional precision, and rounded right-shifting $a\bar{b}$ by r bits exactly corresponds to rounding the fixed-point value to the nearest integer.

In this perspective, it is natural to come up with ways to generalize Barrett's trick: By developing different estimators for the (likely) non-integral value ab/q and/or its nearest integer $\lfloor ab/q \rceil$, different variants of Barrett reduction/multiplication arise, each fitting some specific use-cases. In the following, we depict this idea by presenting some examples employed in our implementation.

Crude Barrett Reduction Cruder approximation will lead to possibly faster reduction but larger output range. When $q = 4591$ (or other number slightly larger than $2^{12} = 4096$), the following is possible. Consider the value $a/4096$. We argue that this is a serviceable approximation for a/q (and for $\lfloor a/q \rceil$ by extension). Quantitatively, this approximation will always overestimate the magnitude by $4591/4096 - 1 \approx 12.1\%$. In view of this, we opted to use $\text{trunc}(a/4096)$ as our estimator, since the truncation or rounding-towards-zero function $\text{trunc}(\cdot)$ helps combat the magnitude overestimation. This gives us Algorithm 2.

With a little effort, the bound $[-4591, 4591]$ on the output is derivable, given that the

doi:10.6342/NTU202501941

input fits in a 16-bit integer, i.e. has the bound $[-32768, 32767]$. Though, by building a model in CryptoLine (or simply enumerating every inputs), we obtain a tighter bound $[-4096, 4096]$. While considerably larger than the bound $[-2881, 2881]$ that standard Barrett reduction guarantees (when using 15-bit right-shifts, conveniently computed with `sqrdmulh`), it is perfectly usable in many cases.

Note that this alternative way of reduction uses three instructions to estimate $\lfloor a/q \rceil$, while standard Barrett uses only one `sqrdmulh` instruction. Consequently, this is only faster when the multiplier pipeline is under heavy pressure. Therefore, in different specific use-cases, one should always benchmark and compare the reduction methods carefully, before deciding on the variant to use.

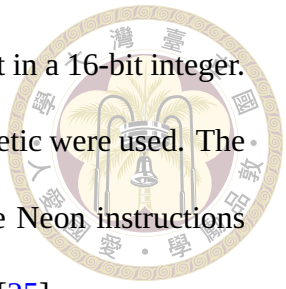
Algorithm 2 Crude Barrett reduction, with specific modulus 4591

Require:
`v0.8h` $= \vec{x}_0$ where $-2^{15} \leq \vec{x}_{0i} \leq 2^{15} - 1$,
precomputed constant stored as `v1.h[0]` $= 4591$
Ensure: `v0.8h` $\equiv \vec{x}_0 \pmod{4591}$ such that $-4096 \leq$ `v0.h[i]` ≤ 4096

```
1: sshr v2.8h, v0.8h, #12
2: cmlt v3.8h, v0.8h, #0
3: sub v2.8h, v2.8h, v3.8h
4: mls v0.8h, v2.8h, v1.h[0]
```

Narrowing Barrett Multiplication At the end of the base-case weighted convolutions, we will have 32-bit intermediate values that need to be reduced to 16-bit ranges, occasionally with a multiply-by-constant step merged together. Since Neon offers a 32-by-32 bit `sqrdmulh` instruction, we can employ Barrett reduction and multiplication for these tasks. With the `uzp1` instruction to extract the low 16-bits of each 32-bit elements, a "narrowing" variant is possible, shown in Algorithm 3 and 4. The key idea is that only the estimation $e = \lfloor a\bar{b}/2^{31} \rceil \approx \lfloor ab/q \rceil$ needs 32-bit arithmetic. After obtaining e, all the remaining computations can be done in 16-bit. This follows from the bound for the original

doi:10.6342/NTU202501941



Barrett multiplication $|ab - eq| \leq q$, which guarantees the output to fit in a 16-bit integer. Consequently, the computed value will be the same as if 32-bit arithmetic were used. The term "narrowing" is in reference to the naming convention for those Neon instructions where the outputs' bit-widths are half of the inputs' [4, Section 3.2.2][25].

Algorithm 3 Narrowing Barrett reduction, with general modulus q

Require:
`v0.4s ++ v1.4s` $= \vec{x}_0$ where $-2^{31} \leq \vec{x}_{0i} \leq 2^{31} - 1$,
precomputed constants stored as `v2.s[0]` $= \lfloor 2^{31}/q \rceil$, `v2.h[2]` $= q$
Ensure: `v3.8h` $\equiv \vec{x}_0 \pmod{q}$ such that $-q \leq$ `v3.h[i]` $\leq q$

```
1: uzp1 v3.8h, v0.8h, v1.8h
2: sqrdmulh v0.4s, v0.4s, v2.s[0]
3: sqrdmulh v1.4s, v1.4s, v2.s[0]
4: uzp1 v4.8h, v0.8h, v1.8h
5: mls v3.8h, v4.8h, v2.h[2]
```

Algorithm 4 Narrowing Barrett multiplication, with general modulus q

Require:
`v0.4s ++ v1.4s` $= \vec{x}_0$ where $-2^{31} \leq \vec{x}_{0i} \leq 2^{31} - 1$,
precomputed constants stored as `v2.s[0]` $= \lfloor 2^{31}b/q \rceil$, `v2.h[2]` $= q$, `v2.h[3]` $= b$
Ensure: `v3.8h` $\equiv b\vec{x}_0 \pmod{q}$ such that $-q \leq$ `v3.h[i]` $\leq q$

```
1: uzp1 v3.8h, v0.8h, v1.8h
2: sqrdmulh v0.4s, v0.4s, v2.s[0]
3: sqrdmulh v1.4s, v1.4s, v2.s[0]
4: uzp1 v4.8h, v0.8h, v1.8h
5: mul v3.8h, v3.8h, v2.h[3]
6: mls v3.8h, v4.8h, v2.h[2]
```


doi:10.6342/NTU202501941

doi:10.6342/NTU202501941

Chapter 4 Verification

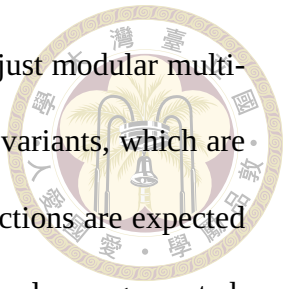
C++ intrinsic functions are used to employ Neon instructions in our implementation (Section 3). Compiler optimization moreover is enabled to reschedule our implementation. With algorithmic optimizations described in the previous section, the correctness of our implementation is not obvious and demands proofs. In order to ensure the correctness of our implementation, we use the CryptoLine tool to formally verify the functional correctness of our implementation.

Concretely, we verify that our compiler-optimized implementation correctly computes the product AB in $\mathbb{Z}_q[x]/\langle x^{761} - x - 1\rangle$ with $q = 4591$ for all input polynomials $A = \sum_{i=0}^{760} a_i x^i$ and $B = \sum_{i=0}^{760} b_i x^i$ with $a_i, b_i \in [-2295, 2295]$ for $0 \leq i < 761$. Our end-to-end verification ensures all ideas in Section 3 are implemented correctly but also correct themselves.

4.1 Correctness and Range Analysis of Modular Arithmetics

Roughly speaking, the modular arithmetics used in the algorithm can be classified into two categories. The characteristic difference between them motivate two slightly different approches, in order to verify them more effectively.

doi:10.6342/NTU202501941



For modular multiplication (and reduction, which is effectively just modular multiplication by 1), these are computed via Barrett multiplication and its variants, which are in turn implemented with multiple instructions. Some of these instructions are expected to overflow, and we need to verify that congruent product (modulo q) is always generated, whether overflow occurred or not. Moreover, bounding the outputs of these arithmetics is non-trivial. Remarkably, both of these properties can be specified in the linear integer arithmetic (LIA) fragment of the CryptoLine language. Equipped with this, we annotate and prescribe both congruential properties and magnitude bounds of each modular multiplications. The CryptoLine tool then proceeds to verify them by invoking the linear integer arithmetic solver isl.

In contrast, addition, subtraction and left-shift (to compute multiplication by 2 or 4) are computed directly with 16-bit `add`, `sub` and `shl` instructions. Here, overflow invariably leads to computation error, thus its absence must be checked with CryptoLine's safety engine. After that, the congruential property and range analysis become rather trivial. As such, while the isl engine can verify them without problems, we often use the algebraic and range fragment of CryptoLine respectively, in order to speed up the verification process. Within the CryptoLine tool, the former is reduced to ideal membership problems and solved by the computer algebra system Singular, while the latter is formulated as Satisfiability Modulo Theories problems (SMT) over the theory of Quantifier-Free Bit-Vectors (QFBV), and solved by the SMT solver Boolector.

doi:10.6342/NTU202501941

4.2 Algebraic Transformations

After ensuring the correctness of modular arithmetics, we are ready to verify the algebraic properties of our algorithm. Recall that given input polynomials $A = \sum_{i=0}^{760} a_i x^i$, $B = \sum_{i=0}^{760} b_i x^i$, the output polynomial $C = \sum_{i=0}^{760} c_i x^i = AB \bmod (x^{761} - x - 1)$ is computed in multiple stages:

Main part Compute $C^{\text{main}} = \sum_{i=0}^{1439} c_i^{\text{main}} x^i = AB \bmod (x^{1440} - 1)$.

Low part Compute $C^{\text{low}} = \sum_{i=0}^{80} c_i^{\text{low}} x^i = AB \bmod x^{81}$.

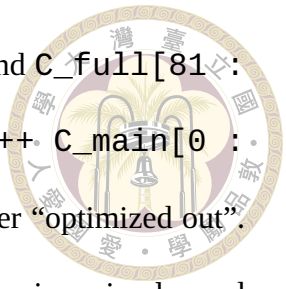
CRT Compute $C^{\text{full}} = \sum_{i=0}^{1520} c_i^{\text{full}} x^i = AB \bmod ((x^{1440} - 1)x^{81}) = AB$ by combining C^{main} and C^{low}. Note that the last equality holds because $\deg(AB) \leq 1520$.

Final Reductions Compute $C = \sum_{i=0}^{760} c_i x^i = C^{\text{full}} \bmod (x^{761} - x - 1) = AB \bmod (x^{761} - x - 1)$.

Ultimately, we are only interested in showing the end-to-end correctness of the algorithm (i.e. $C = AB \bmod (x^{761} - x - 1)$). In a perfect world, CryptoLine should ideally be able to infer this automatically, without detailed knowledge of the algorithm design. In practice, however, we need to speed up the verification process by guiding CryptoLine with cuts and midconditions. In view of this, we decided to additionally verify that the 3 intermediate polynomials $C^{\text{main}}, C^{\text{low}}, C^{\text{full}}$ are computed correctly, alongside the final output polynomial C.

Note that in the actual implementation, there is no dedicated array for storing the coefficients of C^{main} or C^{low}, as mentioned in Section 3.2.4. Nonetheless, before step (c) (as defined in that section), we can reinterpret the `C_full` array as the pair `(C_low,`

doi:10.6342/NTU202501941



`C_main)`. More specifically, `C_full[0 : 81]` stores `C_low`, and `C_full[81 : 1521]` stores the concatenated segments `C_main[81 : 1440] ++ C_main[0 : 81]`. In this sense, we will still treat $C^{\text{main}}, C^{\text{low}}$ as if they were never "optimized out". Indeed, this viewpoint helps us demonstrate our verification strategies in a simpler and more generalizable way.

Remarkably, even though the verification of C^{main} is only a part of our final goal, we are still effectively verifying a full-blown NTT-based polynomial multiplication algorithm, comparable to the previous work where CryptoLine was used to verify various implementations for Kyber, Saber and NTRU [20]. In fact, for each of the implementations, they only showed the correctness of the NTT and iNTT subroutines. In particular, they neither verified the base-case weighted convolutions, nor showed that when strung together, the three subroutines (NTT, weighted convolutions and iNTT) indeed compute polynomial multiplication (in the specific quotient ring) as desired. In contrast, while we still had to treat the subroutines within the main part separately for practical reasons, we also combined the correctness the subroutines to show the correctness the whole main part algorithm.

Because of the incompleteness of the algebraic fragment of CryptoLine, special care has to be taken when writing the postconditions of each cuts, otherwise the Singular engine will fail to deduce the combined correctness. Since this is the first time (to the best of our knowledge) CryptoLine is used to tackle algebraic properties of such complexity, for the sake of future reference, we would like to record the commonly encountered pitfalls and how to circumvent them in general. After that, we will go on and discuss how our implementation—not only the main part but the other 3 stages as well—are verified in more concrete details.

doi:10.6342/NTU202501941

4.2.1 Specifying Equality between Polynomials

For illustration purposes, consider the specific case where we want to specify

$$C^{\text{full}} = AB\,, \tag{4.1}$$

the desired property of C^{full}. There are two ways to denote this in CryptoLine.

Coefficient-wise form When the constituent coefficients of A, B (a_i, b_i for $0 \leq i \leq 760$) and C (c_i for $0 \leq i \leq 1520$) are all available, we can expand RHS and compare coefficients. This results in 1521 coefficient-wise propositions

$$\begin{aligned}
c_0^{\text{full}} &\equiv a_0 b_0 \pmod{q} \\
c_1^{\text{full}} &\equiv a_0 b_1 + a_1 b_0 \pmod{q} \\
&\vdots \\
c_{1519}^{\text{full}} &\equiv a_{759} b_{760} + a_{760} b_{759} \pmod{q} \\
c_{1520}^{\text{full}} &\equiv a_{760} b_{760} \pmod{q}\,.
\end{aligned}$$

These exactly characterise all solutions to Equation 4.1. This coefficient-wise form for writing postconditions can be useful in some cases. For example, one can use multiple cuts to verify the coefficients chunk-by-chunk. Moreover, the validity check for the `eqmod`s can be handled with engines other than Singular. Nonetheless, the sheer length of it make it sometimes difficult to work with.

Polynomial form In CryptoLine, multiple variables can be combined into a polynomial in $\mathbb{Z}[x_1, x_2, \dots]$, where the indeterminates $x_1, x_2, \dots$ are purely formal and do not corre-

doi:10.6342/NTU202501941

spond to any computed values. With this feature, only one proposition is needed instead of 1521:

$$\sum_{i=0}^{1520} c_i^{\text{full}} x^i \equiv \left(\sum_{i=0}^{760} a_i x^i\right)\left(\sum_{i=0}^{760} b_i x^i\right) \pmod{q}. \tag{4.2}$$

Alternatively, we can create new ghost variables for A, B, C^{full} while using the corresponding formal polynomial as their definitions. This way, we can write even more concisely as

$$C^{\text{full}} \equiv AB \pmod{q}. \tag{4.3}$$

Note that in either case, we have to designate the coefficient modulus q explicitly, because CryptoLine always uses $\mathbb{Z}$ as the base ring when invoking Singular.

Incompleteness of Singular To a human, the coefficient-wise form and the polynomial form are clearly equivalent. We thus naturally expect that Singular can deduce one from the other. However, although the "coefficient-wise $\Rightarrow$ polynomial" direction is fine, Singular fails to show the converse. While not technically accurate,[1] an intuitive explanation of this failure is as follows. Because CryptoLine has no way to tell Singular that the coefficients $a_i, b_i, c_i^{\text{full}}$ cannot themselves be polynomials in x, the engine is able to find false-positive counterexamples. For instance, when $a_i, b_i, c_i^{\text{full}}$ are all zeros except $a_0 = 1, b_0 = x, c_1^{\text{full}} = 1$, Equation 4.2 clearly holds but

$$1 = c_1^{\text{full}} \not\equiv a_0 b_1 + a_1 b_0 = 0 \pmod{q}.$$

This is a common pitfall when using the Singular engine to reason about polynomial multiplication, especially when even more formal indeterminates are involved, e.g. during

[1] Singular does not search counterexamples explicitly, and the "formal indeterminate" x is not treated any differently.

doi:10.6342/NTU202501941

verification of the Good-Thomas NTT.

4.2.2 Specifying Congruence between Polynomials

When quotient rings are involved, even more subtleties arise. Consider the property

$$C = AB \bmod (x^{761} - x - 1)\,. \tag{4.4}$$

Note that the algebraic fragment does not provide the binary modulo operator (bmod); we have to replace/simulate it somehow. Depending on the form chosen, this is done differently.

Coefficient-wise form As long as a_i, b_i, c_i for $0 \leq i \leq 760$ are available, the RHS can be expanded again, albeit more tidiously this time. This gives us 761 coefficient-wise propositions:

$$\begin{aligned}
c_0 &\equiv \sum_{\substack{0\leq i,j\leq 760 \\ i+j\in\{0,761\}}} a_i b_j \pmod q \\
c_1 &\equiv \sum_{\substack{0\leq i,j\leq 760 \\ i+j\in\{1,761,762\}}} a_i b_j \pmod q \\
&\vdots \\
c_{759} &\equiv \sum_{\substack{0\leq i,j\leq 760 \\ i+j\in\{759,1519,1520\}}} a_i b_j \pmod q \\
c_{760} &\equiv \sum_{\substack{0\leq i,j\leq 760 \\ i+j\in\{760,1520\}}} a_i b_j \pmod q\,.
\end{aligned}$$

These still exactly characterise all solutions to Equation 4.4. If A, B has higher degree terms, they can be incorporated as well by adding more terms on the RHS.

doi:10.6342/NTU202501941

Polynomial form In CryptoLine, formal polynomials can be included into `eqmod`'s list of modulus beside q. We can therefore write

$$C \equiv AB \pmod{[q, x^{761} - x - 1]}. \tag{4.5}$$

Keep in mind that this proposition is technically weaker than Equation 4.4, since we did not require $\deg(C) \leq 760$ here. Writing $\sum_{i=0}^{760} c_i x^i$ in place of C does not help either: Singular still suspects that c_i might themselves be polynomials in x, e.g. $a_{760} = 1, b_1 = 1, c_{760} = x$ with $A = x^{760}, B = x, C = x^{761}$.

4.3 Main Part

4.3.1 Forward NTT

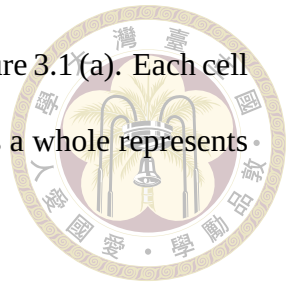
The main challenge in verifying the main part is the 10-by-9 Good-Thomas NTT used to implement 90-NTT. The natural way to think about it is the isomorphism chain

$$\begin{aligned}
\mathbb{Z}_q[x]/\langle x^{1440} - 1\rangle &\cong \mathbb{Z}_q[x, y, z]/\langle x^{16} - yz, y^{10} - 1, z^9 - 1\rangle \\
&\cong \prod_{i=0}^{9} \mathbb{Z}_q[x, y, z]/\langle x^{16} - yz, y - \omega_{10}^i, z^9 - 1\rangle \\
&\cong \prod_{i=0}^{9}\prod_{j=0}^{8} \mathbb{Z}_q[x, y, z]/\langle x^{16} - yz, y - \omega_{10}^i, z - \omega_9^j\rangle \\
&\cong \prod_{i=0}^{9}\prod_{j=0}^{8} \mathbb{Z}_q[x]/\langle x^{16} - \omega_{10}^i\omega_9^j\rangle .
\end{aligned} \tag{4.6}$$

Since the NTT computation is the same for both input A and B, we consider only the former case in the following.

The first isomorphism corresponds to rearranging the $1440 = 90 \times 16$ coefficients of

doi:10.6342/NTU202501941



the zero-padded version of A into a 10-by-9 2D array according to Figure 3.1 (a). Each cell then represents a length-16 polynomial f_k in x, while the 2D array as a whole represents a polynomial in x, y, z, namely

$$\begin{aligned}\Gamma(x,y,z) = \quad & (f_0 + f_{10}z + f_{20}z^2 + \cdots + f_{80}z^8) \\ & + (f_{81} + f_1 z + f_{11}z^2 + \cdots + f_{71}z^8)y \\ & \vdots \\ & + (f_{18} + f_{28}z + f_{38}z^2 + \cdots + f_8 z^8)y^8 \\ & + (f_9 + f_{19}z + f_{29}z^2 + \cdots + f_{89}z^8)y^9 \\ = & \sum_{i=0}^{9}\sum_{j=0}^{8} f_{R(i,j)} y^i z^j \,.\end{aligned}$$

Here the Ruritanian permutation simplifies to $R(i,j) = (81i + 10j) \bmod 90$. Since both the zero-padding and rearrangement are not performed explicitly in the implementation, it is rather meaningless to "verify this isomorphism". One could annotate the expected property that

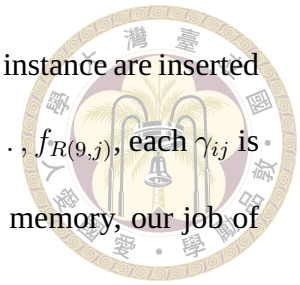
$$\Gamma \equiv A \pmod{[q, x^{16} - yz, y^{10} - 1, z^9 - 1]}$$

explicitly, in the hope that Singular verify the later isomorphisms with greater ease, but we opted not to do this.[2]

The next isomorphism corresponds to the 10-NTT layer (`main_lay1__fwd`). To speed up the verification, we dedicated a cut for this subroutine alone. Consequently, unlike before, we must record the effect of this isomorphism in the cut's postcondition. In

[2]Not only is it unnecessary, defining Γ within CryptoLine is also somewhat cumbersome: For each term $x^k y^i z^j$, one needs to work out the index i' to the optimized-out length-1440 array, while ensuring that the original length-761 array $a_{i'}$ is never actually indexed whenever $i' \geq 761$.

doi:10.6342/NTU202501941



our implementation, the 10 outputs $\gamma_{0j}, \ldots, \gamma_{9j}$ from the j-th 10-NTT instance are inserted into column-j of the 2D array. Note that just like the inputs $f_{R(0,j)}, \ldots, f_{R(9,j)}$, each γ_{ij} is itself a length-16 polynomial in x. Thanks to their orderly layout in memory, our job of defining

$$\begin{aligned}
\Gamma_0^{(1)}(x,z) &= \gamma_{00} + \gamma_{01}z + \cdots + \gamma_{08}z^8 \\
\Gamma_1^{(1)}(x,z) &= \gamma_{10} + \gamma_{11}z + \cdots + \gamma_{18}z^8 \\
&\vdots \\
\Gamma_9^{(1)}(x,z) &= \gamma_{90} + \gamma_{91}z + \cdots + \gamma_{98}z^8
\end{aligned}$$

is made a lot easier. In the postcondition, we thus wrote

$$\Gamma_i^{(1)} \equiv 4A \pmod{[q, x^{16} - yz, y - \omega_{10}^i, z^9 - 1]}. \tag{4.7}$$

The factor 4 comes from that of our 10-NTT design (see Section 3.4.1). Other (i)NTT subroutines all have such factors, which will compound multiplicatively and "enlarge" the factor in the following postconditions.

The 9-NTT layer (`main_lay2__fwd`) computing the third isomorphism is verified similarly. Again, each of the computed polynomials (9 per row, 90 in total) is a length-16 polynomial in x. For the polynomial at row-i column-j, here denoted as $\Gamma_{ij}^{(2)}$, we prescribe the expected

$$\Gamma_{ij}^{(2)} \equiv 8A \pmod{[q, x^{16} - yz, y - \omega_{10}^i, z - \omega_9^j]}. \tag{4.8}$$

In the typical explanation for Good-Thomas NTT, one would use the last isomorphism and change the ring to $\mathbb{Z}_q[x]/\langle x^{16} - \omega_{10}^i \omega_9^j \rangle$. This makes it clear that the base-case

doi:10.6342/NTU202501941

is indeed weighted convolution, and that $\omega_{10}^i \omega_9^j$ is the correct weight to use. In view of this, one might want to write

$$\Gamma_{ij}^{(2)} \equiv 8A \pmod{[q, x^{16} - \omega_{10}^i \omega_9^j]}$$

as the concluding postcondition of the main part NTT. Unfortunately, Singular cannot prove this when only given Equation 4.8. This means that when verifying the following weighted convolution subroutine, the seemingly unnecessary indeterminates y, z must be included in the precondition, and also the postcondition for the same reason.

4.3.2 Weighted Convolution

We are now ready to verify the base-case weighted convolutions (`basemul__main`). Recall that after performing NTT for both input A and B, we have two 10-by-9 arrays of length-16 polynomials $\Gamma_{ij}^{(2)}\Big|_A$, $\Gamma_{ij}^{(2)}\Big|_B$. As noted before, while all polynomials are only in x, we can only use

$$\Gamma_{ij}^{(2)}\Big|_A \equiv 8A \pmod{[q, x^{16} - yz, y - \omega_{10}^i, z - \omega_9^j]} \tag{4.9}$$

$$\Gamma_{ij}^{(2)}\Big|_B \equiv 8B \pmod{[q, x^{16} - yz, y - \omega_{10}^i, z - \omega_9^j]} \tag{4.10}$$

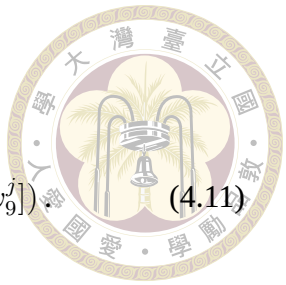
as preconditions here. This is not really an issue: Other than having to write $[q, x^{16} - yz, y - \omega_{10}^i, z - \omega_9^j]$ (instead of $[q, x^{16} - \omega_{10}^i \omega_9^j]$) as the `eqmod`'s list of moduli in the postcondition, the verification process is basically unaffected.

For all $(i, j) \in [0, 9] \times [0, 8]$, there is a dedicated weighted convolution that computes

doi:10.6342/NTU202501941

$\Gamma^{(2)}_{ij}\Big|_{C^{\text{main}}}$ from the two inputs $\Gamma^{(2)}_{ij}\Big|_{A}$, $\Gamma^{(2)}_{ij}\Big|_{B}$. We thus verified that

$$\Gamma^{(2)}_{ij}\Big|_{C^{\text{main}}} \equiv 64AB \pmod{[q, x^{16} - yz, y - \omega_{10}^{i}, z - \omega_{9}^{j}]}. \tag{4.11}$$



4.3.3 Inverse NTT

The inverse NTT essentially goes through the isomorphism chain (Equation 4.6) in reverse. Note that there is nothing to verify for the last isomorphism. The fact that y, z were never eliminated in the postconditions means that we never left the ring $\mathbb{Z}_q[x, y, z]/\langle x^{16} - yz, y - \omega_{10}^{i}, z - \omega_{9}^{j}\rangle$.

The 9-iNTT layer (`main_lay2__bwd`) inverts the second-to-last isomorphism. To confirm this, we verified

$$\Gamma^{(1)}_{i}\Big|_{C^{\text{main}}} \equiv 1152AB \pmod{[q, x^{16} - yz, y - \omega_{10}^{i}, z^{9} - 1]}. \tag{4.12}$$

(Note that $1152 = 64 \cdot 18$.)

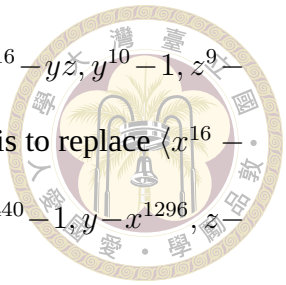
Just like its forward counterpart, the 10-iNTT layer (`main_lay1__bwd`) handles the first two isomorphisms simultaneously. In view of this, we wrote the postcondition in a similar way. Concretely, we skipped the unnecessary

$$\Gamma|_{C^{\text{main}}} \equiv 170AB \pmod{[q, x^{16} - yz, y^{10} - 1, z^{9} - 1]}$$

(where $170 = 1152 \cdot 40 \bmod 4591$), and wrote

$$C^{\text{main}} \equiv 170AB \pmod{[q, x^{16} - yz, y^{10} - 1, z^{9} - 1]}. \tag{4.13}$$

doi:10.6342/NTU202501941



The curious reader might notice that the ring is still $\mathbb{Z}_q[x,y,z]/\langle x^{16}-yz, y^{10}-1, z^9-1\rangle$ and not the desired $\mathbb{Z}_q[x]/\langle x^{1440}-1\rangle$. To solve this, our first step is to replace $\langle x^{16}-yz, y^{10}-1, z^9-1\rangle$ with the identical but differently generated ideal $\langle x^{1440}-1, y-x^{1296}, z-x^{160}\rangle$, giving us

$$C^{\text{main}} \equiv 170AB \pmod{[q, x^{1440}-1, y-x^{1296}, z-x^{160}]}. \tag{4.14}$$

Considering that C^{main} is univariate in x, this readily implies

$$C^{\text{main}} \equiv 170AB \pmod{[q, x^{1440}-1]}. \tag{4.15}$$

Sadly, Singular cannot show this implication. Consequently, we had to manually *but nonetheless soundly* add Equation 4.15 as an assumption. (In the concrete CryptoLine annotation, this is written as an assert/assume pair.) This concludes the verification for the main part algorithm.

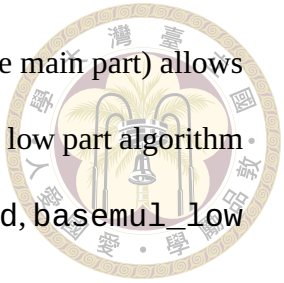
4.4 Low Part

Considering the similarity between the main and low part, it seems that a similar verification strategy would work just as well. However, recall that NTT was used *not* to compute C^{low} but only a part of it, namely the term

$$\left(\sum_{i=0}^{79} a_i x^i\right)\left(\sum_{i=0}^{79} b_i x^i\right) \bmod x^{81} = (A \bmod x^{80})(B \bmod x^{80}) \bmod x^{81}$$

in Equation 3.2. Effectively, the computation for C^{low} was also split into two subtasks, just like C^{full} was. While this approch is still viable, it would require a lot more annotation effort and likely some assert/assume pairs.

doi:10.6342/NTU202501941



Fortunately, the small state space of the low part (compared to the main part) allows us to verify it in an end-to-end manner. In other words, we verified the low part algorithm entirely in one cut, instead of one cut per subroutine (`low_lay1__fwd`, `basemul_low` and `low_lay1__bwd`). We thus wrote

$$C^{\text{low}} \equiv 170AB \pmod{[q, x^{81}]} \tag{4.16}$$

in the postcondition of the only cut of the low part.

4.5 CRT

From C^{main} and C^{low}, we computed the polynomial C^{full} according to Equation 3.1. We verified that indeed

$$C^{\text{full}} \equiv 170AB \pmod{[q, x^{81}]} \tag{4.17}$$

$$C^{\text{full}} \equiv 170AB \pmod{[q, x^{1440} - 1]}, \tag{4.18}$$

and that they do imply

$$C^{\text{full}} \equiv 170AB \pmod{[q, x^{1521} - x^{81}]}. \tag{4.19}$$

However, Singular cannot further show that

$$C^{\text{full}} \equiv 170AB \pmod{q}, \tag{4.20}$$

so we had to add this manually. Together with Equation 4.15, these are the only 2 assert/assume pairs related to polynomial congruence that we included in our annotation. (There

doi:10.6342/NTU202501941

are other assert/assume pairs, which were added to speed up verification of range bounds or coefficient congruence.)

4.6 Final Reductions

The 761 coefficient of C were computed from the 1521 of C^{full} by reducing modulo $x^{761} - x - 1$. The factor of 170 was removed in this step as well. We verified that

$$C \equiv AB \pmod{[q, x^{761} - x - 1]}. \tag{4.21}$$

We also verified that the freezing (see Section 3.2.5) was performed correctly, i.e.

$$c_i \in [-2295, 2295] \tag{4.22}$$

for any $0 \leq i < 761$. (Note that similar range properties are present in all previous cuts. We decided not to include them in the above discussion, just to better focus on the algebraic properties.)

In summary, all newly proposed algebraic transformations indeed work as expected. Moreover, we showed that, barring from the two sound assumptions that were added manually, the Singular engine of CryptoLine is capable of verifying all of the algebraic transformations, either the conventional ones discussed in Section 2 or the novel ones in Section 3.

doi:10.6342/NTU202501941

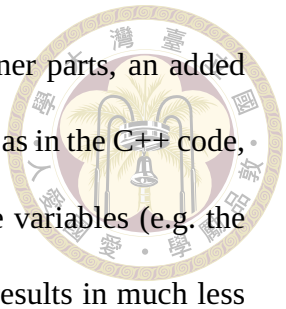
4.7 Compiler Optimization

Each postcondition described above is for its corresponding subroutine as a whole. Most of the time, it's necessary to further partition the subroutine into even more cuts. Take `main_lay1__fwd` as an example. We separated out the 9 instances of 10-NTT acting on each column. One way to do this is to use the coefficient-wise form discussed in Section 4.2. Since we are proving the coefficient-wise form first, prior to combining them in the concluding postcondition for the subroutine, the incompleteness of Singular does not take effect here.

In practice, unfortunately, this was still too coarse-grained to be verified efficiently. We realized the need to partition the 10-NTT further, where the first step would be to separate out one of the 5-NTT (see Section 3.4.1). However, instructions are often reordered when more aggresive compiler optimizations are enabled. In fact, for the binary compiled with the `-O3` setting, the reordering was so aggresive that part of the computation for the second 5-NTT got interleaved with the first 5-NTT. Consequently, we had to prescribe the effect of the included part of the second 5-NTT in the first cut, otherwise we would not be able to verify the second 5-NTT in the next cut, as computation done in the first cut is invisible from the second cut.

In view of this, we decided to apply the technique in [21]. We extracted the same subroutine from the recompiled binary with only the `-O1` flag, verified the less optimized version, then finally verified the equivalence between the two versions. Note that the equivalence checking was not for the whole algorithm, but per subroutine. More specifically, for each of the eight subroutines named in this paper, we had to perform equivalence checking once.

doi:10.6342/NTU202501941



Aside from the ability to cleanly partition each 10-NTT into finer parts, an added bonus is that since the instructions now stay in roughly the same order as in the C++ code, it is much easier to figure out the mapping between the intermediate variables (e.g. the nodes in the butterfly graphs) and the vector registers. This in turn results in much less human-time spent on annotating midconditions.

To summarize, for each of the eight subroutines, we built a CryptoLine model from the recompiled binary. This model was much easier to annotate and verify. Regardless, by finally showing the functional equivalence between the `O1` and the `O3` version, i.e. they produce identical output coefficients given any input, the `O3` version is verified just the same. Using the strategies detailed in the previous sections, we were able to combine the correctness of the subroutines and ultimately show the correctness of the whole polynomial multiplier, specifically the compiled binary instead of the C++ code.

doi:10.6342/NTU202501941

doi:10.6342/NTU202501941

Chapter 5 Results

We evaluate our implementation and report our verification of the optimized polynomial multiplication for NTRU Prime in this section.

5.1 Performance of Polynomial Multiplication

The performance of our NTRU Prime polynomial multiplication is evaluated on Raspberry Pi 3, Pi 4, and Pi 5 with ARM Cortex-A53, Cortex-A72, and Cortex-A76 respectively. We use `gcc 12.2.0` to compile our C++ implementation with the optimization flag `-O3`. Table 5.1 compares our new algorithm with the state of the art in [17] by CPU cycles.

Table 5.1: Cycle count for multiplication in $\mathbb{Z}_q[x]/\langle x^{761} - x - 1\rangle$

Benchmark Environment	[17]	Ours	Speedup
Cortex-A53	50689	39018	1.30x
Cortex-A72	31987	26772	1.19x
Cortex-A76	26295	21165	1.24x

Our evaluation shows that our optimized polynomial multiplication is faster than [17] by at least 19%. On Raspberry Pi 3, the speedup of 30% is attained. Our NTT-based multiplications over a polynomial ring with an atypical factor x^m can be significantly faster than the best prior algorithm for `sntrup761`.

doi:10.6342/NTU202501941

Table 5.2: Cycle counts per subroutine, on Cortex-A72

Subroutine	#Calls	Cycles	Subroutine	#Calls	Cycles
`mainmul`	1	23051	`lowmul`	1	1741
`main_lay1__fwd`	2	2610	`low_lay1__fwd`	2	164
`main_lay2__fwd`	2	2345			
`basemul__main`	1	8142	`basemul__low`	1	1173
`main_lay2__bwd`	1	2358	`low_lay1__bwd`	1	214
`main_lay1__bwd`	1	2619			
`mod_poly`	1	989	`freeze`	1	967

To identify the bottleneck of the polynomial multiplication, Table 5.2 gives the cycle counts for each subroutine on Raspberry Pi 4. The column *#Calls* indicates how many times the subroutine is invoked; the column *Cycles* shows the cycle counts per invocation. As expected, the main part takes about 86% of the total cycles (26772). About 30% of the time is used on the base-case convolutions in the main part.

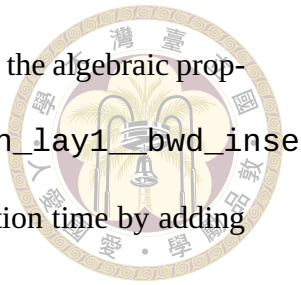
5.2 Cost of Verification

Table 5.3: Verification time for the correctness of `-O1` models, in seconds

Function	Safety	RA	RS	AA	AS
`basemul__low`	48.27	0.03	14.20	56.36	49.78
`basemul__main`	584.40	4.30	491.09	1290.12	1923.39
`low_lay1__bwd`	632.69	1028.02	283.11	3.12	1.92
`low_lay1__fwd`	91.67	177.31	4.50	2.47	2.60
`main_lay1__bwd`	271.28	100.63	122.85	3460.13	267.00
`main_lay1__fwd`	1086.61	2.73	1236.63	684.17	191.31
`main_lay2__bwd`	1902.30	0.49	156.55	777.60	177.99
`main_lay2__fwd`	1800.14	0.62	187.47	4491.65	223.34

Table 5.3 details the verification time for the polynomial multiplication in seconds. The column *Safety* shows the time for overflow checking. The columns *RA* (range assertion) and *RS* (range specification) give the time for range analysis. And *AA* (algebraic assertion) and *AS* (algebraic specification) show the time for algebraic property verification.

doi:10.6342/NTU202501941



Verification of the main part occupies a large portion of time. Notably, the algebraic properties in 10-NTT (`main_lay2__fwd_inplace`) and 9-iNTT (`main_lay1__bwd_insert`) take more than 50 minutes to verify. One could improve the verification time by adding more proof hints manually [20]. We leave it as a future work.

Our equivalence checking calls the `abc` toolkit through CryptoLine. Note that between the optimization flags `-O1` and `-O3`, the `gcc` compiler does a lot of complex transformations and rescheduling for code generation, which explains the week-long wall time there. However, equivalence checking is mostly automatic and requires little human guidance. Manually adding more hints can reduce the verification time and is left as a future work.

Table 5.4: Verification time for the equivalence between `-O3` and `-O1` models, in seconds

Function	Equivalence	Function	Equivalence
`basemul__low`	586994.63	`main_lay1__bwd`	911.66
`basemul__main`	3642.91	`main_lay1__fwd`	829.96
`low_lay1__bwd`	24.81	`main_lay2__bwd`	835.79
`low_lay1__fwd`	26.52	`main_lay2__fwd`	827.69

doi:10.6342/NTU202501941

doi:10.6342/NTU202501941

Chapter 6 Conclusion

We have described how one obtain a high-performance, high-assurance polynomial multiplication useful for NTRU Prime. We hope that the detailed description also serves as a documentation for our code and an instruction instrument as well as an engineering manual for people interested in unorthodox NTT techniques.

For future work, we can still consider various possible optimizations; one possibility is adapting the radix-3 butterfly used in [15, 19]. We can also adapt the optimizations to other NTRU variant cryptosystems.

doi:10.6342/NTU202501941

doi:10.6342/NTU202501941

References

[1] E. Alkim, D. Y.-L. Cheng, C.-M. M. Chung, H. Evkan, L. W.-L. Huang, V. Hwang, C.-L. T. Li, R. Niederhagen, C.-J. Shih, J. Wälde, and B.-Y. Yang. Polynomial multiplication in NTRU prime. *IACR TCHES*, 2021(1):217–238, 2021.

[2] J. B. Almeida, M. Barbosa, G. Barthe, B. Grégoire, V. Laporte, J.-C. Léchenet, T. Oliveira, H. Pacheco, M. Quaresma, P. Schwabe, A. Séré, and P.-Y. Strub. Formally verifying Kyber episode IV: Implementation correctness. *IACR TCHES*, 2023(3):164–193, 2023.

[3] J. B. Almeida, S. A. Olmos, M. Barbosa, G. Barthe, F. Dupressoir, B. Grégoire, V. Laporte, J.-C. Léchenet, C. Low, T. Oliveira, H. Pacheco, M. Quaresma, P. Schwabe, and P.-Y. Strub. Formally verifying kyber - episode V: Machine-checked IND-CCA security and correctness of ML-KEM in EasyCrypt. In L. Reyzin and D. Stebila, editors, *CRYPTO 2024, Part II*, volume 14921 of *LNCS*, pages 384–421. Springer, Cham, Aug. 2024.

[4] ARM. *NEON Programmer's Guide*, 2013. https://developer.arm.com/documentation/den0018/a.

[5] M. Barbosa, G. Barthe, X. Fan, B. Grégoire, S.-H. Hung, J. Katz, P.-Y. Strub, X. Wu,

doi:10.6342/NTU202501941

and L. Zhou. EasyPQC: Verifying post-quantum cryptography. In G. Vigna and E. Shi, editors, *ACM CCS 2021*, pages 2564–2586. ACM Press, Nov. 2021.

[6] H. Becker, V. Hwang, M. J. Kannwischer, B.-Y. Yang, and S.-Y. Yang. Neon NTT: Faster Dilithium, Kyber, and Saber on Cortex-A72 and Apple M1. *IACR TCHES*, 2022(1):221–244, 2022.

[7] D. J. Bernstein. Multidigit multiplication for mathematicians. 2001. https://cr.yp.to/papers.html#m3.

[8] D. J. Bernstein, B. B. Brumley, M.-S. Chen, C. Chuengsatiansup, T. Lange, A. Marotzke, B.-Y. Peng, N. Tuveri, C. van Vredendaal, and B.-Y. Yang. NTRU Prime. Technical report, National Institute of Standards and Technology, 2020. available at https://csrc.nist.gov/projects/post-quantum-cryptography/post-quantum-cryptography-standardization/round-3-submissions.

[9] D. J. Bernstein, B. B. Brumley, M.-S. Chen, and N. Tuveri. OpenSSLNTRU: Faster post-quantum TLS key exchange. In K. R. B. Butler and K. Thomas, editors, *USENIX Security 2022*, pages 845–862. USENIX Association, Aug. 2022.

[10] H.-T. Chen, Y.-H. Chung, V. Hwang, and B.-Y. Yang. Algorithmic views of vectorized polynomial multipliers - NTRU. In A. Chattopadhyay, S. Bhasin, S. Picek, and C. Rebeiro, editors, *INDOCRYPT 2023, Part II*, volume 14460 of *LNCS*, pages 177–196. Springer, Cham, Dec. 2023.

[11] C.-M. M. Chung, V. Hwang, M. J. Kannwischer, G. Seiler, C.-J. Shih, and B.-Y.

doi:10.6342/NTU202501941

Yang. NTT multiplication for NTT-unfriendly rings. *IACR TCHES*, 2021(2):159–188, 2021.

[12] R. Crandall and B. Fagin. Discrete weighted transforms and large-integer arithmetic. *Math. Comput.*, 62(205):305–324, Jan. 1994.

[13] E. Fujisaki and T. Okamoto. Secure integration of asymmetric and symmetric encryption schemes. In M. J. Wiener, editor, *CRYPTO'99*, volume 1666 of *LNCS*, pages 537–554. Springer, Berlin, Heidelberg, Aug. 1999.

[14] I. J. Good. The interaction algorithm and practical fourier analysis. *J. of the Royal Statistical Society, Series B*, 20(2):361–372, 1958.

[15] C. A. Hassan and O. Yayla. Radix-3 NTT-based polynomial multiplication for lattice-based cryptography. Cryptology ePrint Archive, Report 2022/726, 2022.

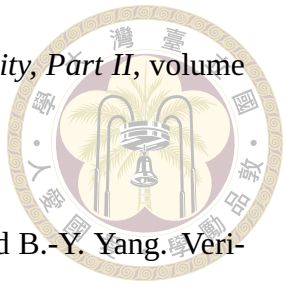
[16] J. Huang, J. Zhang, H. Zhao, Z. Liu, R. C. C. Cheung, Ç. K. Koç, and D. Chen. Improved plantard arithmetic for lattice-based cryptography. *IACR TCHES*, 2022(4):614–636, 2022.

[17] V. Hwang. Pushing the limit of vectorized polynomial multiplications for NTRU prime. In Y. L. Tianqing Zhu, editor, *ACISP 24, Part II*, volume 14896 of *LNCS*, pages 84–102. Springer, Singapore, July 2024.

[18] V. Hwang, Y. Kim, and S. C. Seo. Barrett multiplication for dilithium on embedded devices. Cryptology ePrint Archive, Report 2023/1955, 2023.

[19] V. Hwang, C.-T. Liu, and B.-Y. Yang. Algorithmic views of vectorized polynomial multipliers - NTRU prime. In C. Pöpper and L. Batina, editors, *ACNS 24Interna-*

doi:10.6342/NTU202501941



tional Conference on Applied Cryptography and Network Security, Part II, volume 14584 of *LNCS*, pages 24–46. Springer, Cham, Mar. 2024.

[20] V. Hwang, J. Liu, G. Seiler, X. Shi, M.-H. Tsai, B.-Y. Wang, and B.-Y. Yang. Verified NTT multiplications for NISTPQC KEM lattice finalists: Kyber, SABER, and NTRU. *IACR TCHES*, 2022(4):718–750, 2022.

[21] L.-C. Lai, J. Liu, X. Shi, M.-H. Tsai, B.-Y. Wang, and B.-Y. Yang. Automatic verification of cryptographic block function implementations with logical equivalence checking. In J. Garcia-Alfaro, R. Kozik, M. Choraś, and S. Katsikas, editors, *ESORICS 2024, Part IV*, volume 14985 of *LNCS*, pages 377–395. Springer, Cham, Sept. 2024.

[22] C. M. Rader. Discrete fourier transforms when the number of data samples is prime. *Proceedings of the IEEE*, 56(6):1107–1108, 1968.

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

[24] L. H. Thomas. *Applications of Digital Computers*, chapter Using a computer to solve problems in physics. Ginn, Boston, 1963.

[25] Y. Zhang. ARM NEON programming quick reference, 2015. https://community.arm.com/arm-community-blogs/b/operating-systems-blog/posts/arm-neon-programming-quick-reference.

doi:10.6342/NTU202501941