國立臺灣大學電資學院電機工程學系

碩士論文

Department of Electrical Engineering

College of Electrical Engineering and Computer Science

National Taiwan University

Master Thesis

# 在 ARM 上實作晶格密碼系統中之多項式模反元素計算

# Implementation of Polynomial Modular Inversion in Lattice-based cryptography on ARM

李菁琳

Ching-Lin Li

指導教授: 黃俊郎 博士

Advisor: Jiun-Lang Huang Ph.D.

中華民國 110 年 9 月

September, 2021

doi:10.5566/NTU2021XXXXX

# 國立臺灣大學碩士學位論文

## 口試委員會審定書

在 ARM 上實作晶格密碼系統中之
多項式模反元素計算

Implementation of Polynomial Modular Inversion in Lattice-based cryptography on ARM

本論文係李菁琳君（R08921A15）在國立臺灣大學電機工程學系資訊安全碩士班完成之碩士學位論文，於民國 110 年 9 月 1 日承下列考試委員審查通過及口試及格，特此證明

口試委員：

（指導教授）

所　　長：

doi:10.5566/NTU2021XXXXX

# Acknowledgements

初來到實驗室，對於研究主題及方向一竅不通，感謝當時鄭老師提供給我一些方向，也感謝黃老師在後續對我們的照顧及提點。碩一下後，我才逐漸了解自己學習的演算法對於提升後量子密碼系統有著顯著的影響。在學習及研究的過程中，常常遇到挫折，不論是卡住或是實作成果不如預期時，楊老師總是不厭其煩地提供新的實作技巧及想法，也逐步解釋我搞不懂的地方，特別感謝楊老師的耐心指導。除此之外，也特別感謝實驗室的學長姐及同學們，在學習的路上提供許多的幫助，讓我能減少鬼打牆的時間，順利完成學業。最後，感謝我的家人們，給予我關懷與支持，讓我能沒有後顧之憂。由衷的感謝大家！

doi:10.5566/NTU2021XXXXX

# 摘要

美國國家標準暨技術研究院於近年舉辦後量子密碼學標準化競賽以抵禦未來量子電腦帶來的資安問題。NTRU 及 Streamlined NTRU Prime 各別為進入最終階段的入選者及候補者。本篇論文將針對其在 ARM Cortex-M4 平台上金鑰生成的效能進行優化，影響其效能之關鍵瓶頸為多項式模反元素的計算。本實作使用一個歐幾里德算法的流線型恆定時間變體來優化金鑰生成的速度，NTRU 的每組參數皆比當前最佳的實作快 96% 至 97%，而 Streamlined NTRU Prime 的參數中，除了 sntrup761 之外，皆快了約 93%。

**關鍵字：**後量子密碼學、晶格密碼系統、NTRU、NTRU Prime、多項式模反元素、輾轉相除法、ARM Cortex-M4 實作

doi:10.5566/NTU2021XXXXX

# Abstract

The National Institute of Standards and Technology has held a post-quantum cryptography standardization competition in recent years against the security problems brought by quantum computers in the future. NTRU is a candidate; Streamlined NTRU Prime is a alternate. This thesis will optimize the performance of key generation of both cryptosystems on the ARM Cortex-M4 platform. The key bottleneck of their performance is the calculation of the polynomial modular reciprocal. This work uses a streamlined constant-time variant of Euclid's algorithm to optimize the speed of key generation. Each parameter set of NTRU is between 96% and 97% faster than the current best implementation; the parameter sets of Streamlined NTRU Prime, except for sntrup761, are all about 93% faster than the current best implementation.

**Keywords:** Post-quantum cryptography, Lattice-based cryptography, NTRU, NTRU Prime, polynomial reciprocal, Euclid's algorithm, ARM Cortex-M4 implementation

doi:10.5566/NTU2021XXXXX

# Contents

doi:10.5566/NTU2021XXXXX

doi:10.5566/NTU2021XXXXX

# List of Figures

doi:10.5566/NTU2021XXXXX

# List of Tables

doi:10.5566/NTU2021XXXXX

doi:10.5566/NTU2021XXXXX

# Denotation

| | |
|---|---|
| $\Phi_1$ | $(x-1)$ |
| $\Phi_n$ | $(x^n-1)/(x-1) = x^{n-1} + x^{n-2} + \cdots + x + 1, n > 1$ |
| $R_{(NTRU)}$ | $Z[x]/(\Phi_1\Phi_n)$, a polynomial ring used in NTRU |
| $R_{q(NTRU)}$ | $(Z/q)[x]/(\Phi_1\Phi_n)$ where $q$ is a positive integer, a polynomial ring used in NTRU |
| $S_2$ | $(Z/2)[x]/(\Phi_n)$, a polynomial ring used in NTRU |
| $S_3$ | $(Z/3)[x]/(\Phi_n)$, a polynomial ring used in NTRU |
| $R$ | $Z[x]/(x^p - x - 1)$ where $p$ is a prime number, a polynomial ring used in Streamlined NTRU Prime |
| $R_q$ | $(Z/q)[x]/(x^p - x - 1)$ where $q$ and $p$ are prime numbers, a polynomial ring used in Streamlined NTRU Prime |
| $deg(f)$ | The degree of the polynomial $f$. |

doi:10.5566/NTU2021XXXXX

# Chapter 1 Introduction

Recent studies have pointed out that quantum computers can solve NP-hardness problems which are the security core of current cryptosystems. They will destroy the confidentiality and integrity of data in the foreseeable future. Owing to this concern, the National Institute of Standards and Technology (NIST) initiated a competition called the PQC project to call for proposals of public-key post-quantum cryptographic algorithms in 2016. These algorithms must be secure against attacks from computers of the two types and be compatible with current communication protocols. NIST selects algorithms on factors such as security, cost, performance, etc. After three rounds of elimination, they ultimately announced the finalists of candidates and alternates on July 22, 2020. Among public-key encryption and key-establishment algorithms, NTRU becomes one of the promising candidates; NTRU Prime, a variant of NTRU, gains a position of the alternates.

ARM Cortex-M4 microcontroller is one of the platforms evaluated by NIST. The pqm4 project [1] contains the state-of-the-art implementations of the candidates on this platform. Both NTRU and NTRU Prime use lattice-based cryptography. In lattice-based schemes, polynomial operations over different rings are the most time-consuming. Many researchers have tried to use different algorithms, such as Karatsuba and Toom[2], number-theoretic transforms (NTT)[3], etc., to optimize polynomial multiplications. Currently, the operation of polynomial modular reciprocal takes up almost all the running time of

doi:10.5566/NTU2021XXXXX

the NTRU key generation. Streamlined NTRU Prime [4], a type of NTRU Prime scheme, has the same problem. Hence, this thesis aims to further optimize the performance of the polynomial inversion.

## 1.1 Motivation

Previous work of NTRU uses the almost inverse algorithm to implement polynomial inversion. However, the streamlined constant-time variant of Euclid's algorithm can provide a more efficient implementation [5]. The previous work of Streamlined NTRU Prime, except sntrup761, only uses the iteration way to implement the efficient algorithm. The sntrup761 has been only optimized from the aspect of polynomial multiplication in polynomial inversion. Nevertheless, it does not utilize the advantage of the NTT method and the bit slicing technique. Improvements based on these shortcomings should be able to break the primary bottleneck in the key generation to make the two cryptosystems more suitable for the new cryptography standards.

## 1.2 Contribution

This thesis presents two architecture to implement the variant of Euclid's algorithm. It also evaluates the performance of various multiplication methods used in gcd computations. In the end, it provides the fastest solutions for different polynomial quotient rings and applies them to all parameter sets of NTRU and Streamlined NTRU Prime. The performance of the key generation has improved remarkably. The other cryptosystems could reuse this work to implement fast polynomial inversion.

doi:10.5566/NTU2021XXXXX

# Chapter 2 Background

The **pqm4** project provides a framework to test and set a benchmark on the ARM Cortex-M4[1]. Commit b4c013e is state-of-the-art on Jul 23, 2021. All the codes of NTRU and Streamlined NTRU Prime are available at https://github.com/mupq/pqm4/tree/master/crypto_kem. There are four implementations of NTRU based on different parameter settings: **ntruhps2048509**, **ntruhps2048677**, **ntruhps4096821**, and **ntruhrss701**. On the other hand, Streamlined NTRU Prime has six parameter sets denoted $sntrup[p]$, such as **sntrup761**. The subsequent sections will describe these parameter settings and the relation to the polynomial inversions.

## 2.1 NTRU

The NTRU scheme was first devised in 1996 by Hoffstein, Pipher, and Silverman [6]. After continuous evolution and optimization, researchers proposed several variants of this scheme. At present, the latest NTRU submission to the NIST PQC project uses a correct deterministic public-key encryption scheme (DPKE). There are two narrowly defined families of parameter sets: NTRU-HPS and NTRU-HRSS. Despite the different restrictions of parameters in detail, both of them are parameterized by coprime positive integers: $n$, $p$, and $q$; sets of integer polynomials for sample: $\mathcal{L}_f$, $\mathcal{L}_g$, $\mathcal{L}_r$, and $\mathcal{L}_m$; and

doi:10.5566/NTU2021XXXXX

an injection $Lift$: $\mathcal{L}_m \rightarrow Z[x]$[7]. The parameters $n$, $p$, and $q$ determine the quotient rings used in NTRU key generation. Table 2.1 shows all recommended parameter sets of NTRU.

| | $n$ | $q$ | $S_3$ | $S_q$ |
|---|---|---|---|---|
| **ntruhps2048509** | 509 | 2048 | $(Z/3)[x]/(\Phi_{509})$ | $(Z/2048)[x]/(\Phi_{509})$ |
| **ntruhps2048677** | 677 | 2048 | $(Z/3)[x]/(\Phi_{677})$ | $(Z/2048)[x]/(\Phi_{677})$ |
| **ntruhps4096821** | 821 | 4096 | $(Z/3)[x]/(\Phi_{821})$ | $(Z/4096)[x]/(\Phi_{821})$ |
| **ntruhrss701** | 701 | 8192 | $(Z/3)[x]/(\Phi_{701})$ | $(Z/8192)[x]/(\Phi_{701})$ |

Table 2.1: All Recommended Parameter Sets of NTRU

The primary bottleneck of key generation is the three polynomial inversions defined in Algorithm 1. The research problem can be simplified to optimize polynomial inversions over the two quotient rings $S_3$ and $S_q$.

**Algorithm 1** NTRU Key Generation of the DPKE

```
1: $f \leftarrow$ Generate a random polynomial $\in \mathcal{L}_f$
2: $g \leftarrow$ Generate a random polynomial $\in \mathcal{L}_g$
3: $f_p \leftarrow 1/f$ in $S_3$
4: $f_q \leftarrow 1/f$ in $S_q$
5: $h \leftarrow 3 * g * f_q$ in $R_{q(NTRU)}$
6: $h_q \leftarrow 1/h$ in $S_q$
7: **return** $((f, f_p, h_q), h)$
```

## 2.2 Streamlined NTRU Prime

Streamlined NTRU Prime is designed for IND-CCA2 security and is optimized from an implementation aspect. It is a member of the NTRU Prime family having three parameters $(p, q, w)$ and rings of the form $(Z/q)[x]/(x^p - x - 1)$. A Streamlined NTRU Prime parameter set satisfies the following restrictions: both $p$ and $q$ are prime numbers; $w$ is a positive integer; $2p \geq 3w$; $q \geq 16w + 1$; and $(x^p - x - 1)$ is irreducible in the polynomial ring $(Z/q)[x]$[8].

Similar to NTRU, the three polynomial inversion operations in the key generation of

doi:10.5566/NTU2021XXXXX

|  | $p$ | $q$ | $R_3$ | $R_q$ |
|---|---|---|---|---|
| **sntrup653** | 653 | 4621 | $(Z/3)[x]/(x^{653}-x-1)$ | $(Z/4621)[x]/(x^{653}-x-1)$ |
| **sntrup761** | 761 | 4591 | $(Z/3)[x]/(x^{761}-x-1)$ | $(Z/4591)[x]/(x^{761}-x-1)$ |
| **sntrup857** | 857 | 5167 | $(Z/3)[x]/(x^{857}-x-1)$ | $(Z/5167)[x]/(x^{857}-x-1)$ |
| **sntrup953** | 953 | 6343 | $(Z/3)[x]/(x^{953}-x-1)$ | $(Z/6343)[x]/(x^{953}-x-1)$ |
| **sntrup1013** | 1013 | 7177 | $(Z/3)[x]/(x^{1013}-x-1)$ | $(Z/7177)[x]/(x^{1013}-x-1)$ |
| **sntrup1277** | 1277 | 7879 | $(Z/3)[x]/(x^{1277}-x-1)$ | $(Z/7879)[x]/(x^{1277}-x-1)$ |

Table 2.2: All Recommended Parameter Sets of Streamlined NTRU Prime

Streamlined NTRU Prime are the main point to speed up performance. The optimization of polynomial inversions over $R_3$ and $R_q$ is the major research focus.

**Algorithm 2** Streamlined NTRU Prime Core Key Generation

```
1: repeat
2:     $g \leftarrow$ Generate a uniform random small polynomial $\in R$
3: until $g$ is invertible in $R_3$
4: $f \leftarrow$ Generate a uniform random polynomial $\in Short$
5: $ginv \leftarrow 1/g$ in $R_3$
6: $h \leftarrow g/(3f)$ in $R_q$
7: return $(h, (f, ginv))$
```

## 2.3 Constant-Time Polynomial Modular Inversion

From past to now, scholars develop a variant of algorithms to compute modular reciprocals, such as Extended Euclid's algorithm, Stein's gcd algorithm, etc. Despite the properties of efficiency and correctness, many of them are insecure when used in cryptosystems. The attacker may collect secret information through timing attacks; hence, choose a constant-time algorithm to calculate inversion is vital.

Current post-quantum cryptosystems ordinarily use Fermat's little theorem or variants of Euclid's algorithm against timing attacks. The latter defeats the former at performance when the input is large enough. For the polynomial inversions among different parameters in NTRU and NTRU Prime, the minimum degree of the polynomials is 508. Each coefficient of polynomials would be stored in 4 bits, 8 bits, or 16 bits. In other words,

doi:10.5566/NTU2021XXXXX

the inputs all have at least 2000 bits. Consequently, variants of Euclid's algorithm would be a sensible choice for NTRU and NTRU Prime. For now, the fast constant-time modular inversion algorithm is the **division steps** proposed in 2019 [5]. The following introduces the definition of the division steps and the mechanism to get the modular reciprocals after a fixed number of division steps iteration.

### 2.3.1 Divstep

A division step takes 3 parameters $(\delta, f, g)$ where $\delta$ models $deg(f) - deg(g)$, because we are reversing the polynomials. $f(0) \neq 0$.

$$divstep(\delta, f, g) = \begin{cases} (1-\delta, g, (g(0)f - f(0)g)/x) & \text{if } \delta > 0 \text{ and } g(0) \neq 0 \\ (1+\delta, f, (f(0)g - g(0)f)/x) & \text{otherwise.} \end{cases}$$

$(\delta_n, f_n, g_n) = divstep^n(\delta, f, g)$ represents the result of $n$ iterations of the division step. The transition between each iteration can be turned into a matrix $\mathcal{T}$:

$$\begin{pmatrix} f_i \\ g_i \end{pmatrix} = \mathcal{T}_{i-1}(\delta_{i-1}, f_{i-1}, g_{i-1}) \begin{pmatrix} f_{i-1} \\ g_{i-1} \end{pmatrix}$$

where $\mathcal{T}$ is defined as follow:

$$\mathcal{T}_{i-1}(\delta_{i-1}, f_{i-1}, g_{i-1}) = \begin{cases} \begin{pmatrix} 0 & 1 \\ \frac{g_{i-1}(0)}{x} & \frac{-f_{i-1}(0)}{x} \end{pmatrix} & \text{if } \delta > 0 \text{ and } g(0) \neq 0 \\ \begin{pmatrix} 1 & 0 \\ \frac{-g_{i-1}(0)}{x} & \frac{f_{i-1}(0)}{x} \end{pmatrix} & \text{otherwise.} \end{cases}$$

doi:10.5566/NTU2021XXXXX

Then, the computation process of $n$ iterations can be simplified to only one transition matrix product $P$ with four entries $u_n, v_n, r_n, s_n$:

$$\begin{pmatrix} f_n \\ \\ g_n \end{pmatrix} = \mathcal{T}_{n-1}\mathcal{T}_{n-2}...\mathcal{T}_0 \begin{pmatrix} f \\ \\ g \end{pmatrix} = \begin{pmatrix} \frac{u_n}{x^{n-1}} & \frac{v_n}{x^{n-1}} \\ \\ \frac{r_n}{x^n} & \frac{s_n}{x^n} \end{pmatrix} \begin{pmatrix} f \\ \\ g \end{pmatrix}.$$

At each iteration, current $f_i(0)$ and $g_i(0)$ determine a new transition matrix $\mathcal{T}_i$. The four entries, $f_i$, and $g_i$ would be updated by matrix multiplications. Algorithm 3 shows the way to get $f_n$, $f_g$, and $P$.

**Algorithm 3** $divstep^n(\delta, f, g)$

```
1:  $u \leftarrow 1$
2:  $v \leftarrow 0$
3:  $r \leftarrow 0$
4:  $s \leftarrow 1$
5:  while $n > 0$ do
6:      if $\delta > 0$ and $g(0) \,! = 0$ then
7:          $\delta \leftarrow -\delta$
8:          $swap(f, g)$
9:          $swap(u, r)$
10:         $swap(v, s)$
11:     end if
12:     $\delta \leftarrow \delta + 1$
13:     $g \leftarrow (f(0)g - g(0)f)/x$
14:     $r \leftarrow (f(0)r - g(0)u)$
15:     $s \leftarrow (f(0)s - g(0)v)$
16:     $n \leftarrow n - 1$
17:     if $n > 0$ then
18:         $u \leftarrow ux$
19:         $v \leftarrow vx$
20:     end if
21: end while
22: return $(\delta, f, g, u, v, r, s)$
```

$v_n$ and $f_n$ are the keys to obtain the modular reciprocal. Let $d \in Z$ and $d > 0$; $F, G$ are polynomials with $deg(F) = d > deg(G)$; $gcd(F, G) = 1$. Define $f$ as the reversal of $F$, and $g$ is the reversal of $G$. When $n \geq deg(F) + deg(G) = 2d - 1$, $f_n$ would be a

doi:10.5566/NTU2021XXXXX

constant number $k \in Z$. Then, the equation

$$\frac{u_n}{x^{n-1}} f + \frac{v_n}{x^{n-1}} g = k$$

reveals the modular reciprocal $V$ such that $VG \equiv 1 \ (mod \ F)$:

$$V = x^{n-d} v_n(1/x) \ / \ k.$$

Algorithm 4 presents the whole process to obtain the reciprocal of $G$ modulo $F$. The function $reverse(f, d) = x^d f(1/x)$ is used to reverse a polynomial $f$ with degree $d$.

**Algorithm 4** Reciprocal of $G$ Modulo $F$

```
1: $\delta \leftarrow 1$
2: $f \leftarrow reverse(F, d)$
3: $g \leftarrow reverse(G, d-1)$
4: $(\delta_n, f_n, g_n, u_n, v_n, r_n, s_n) \leftarrow divstep^n(\delta, f, g)$
5: if $\delta_n \ != n - (2d-1)$ then
6:     return $null$
7: end if
8: $V \leftarrow reverse(v_n / f_n(0), n-d)$
9: return $V$
```

### 2.3.2 Jumpdivsteps

The feature of the transition matrices can accelerate the speed of $divstep^n(\delta, f, g)$ by using a divide-and-conquer algorithm. A $jumpdivsteps^n$ function outputs the same results as the $divstep^n$ with the same inputs. However, it breaks down the n size problem into two or multiple smaller sub-problems instead of running $n$ iterations of the division step.

The following illustrates a scheme to solve an n size problem by solving two sub-problems with $j$ and $(n-j)$ size, respectively. It first jumps $j$ division steps from an

doi:10.5566/NTU2021XXXXX

initial state to $(\delta_j, f'_j, g'_j)$ and gets the $j$-step transition matrix $M_1 = \mathcal{T}_{j-1}...\mathcal{T}_0$. Notice that it uses only the first $j$ coefficients of inputs, namely $f_L$ and $g_L$. Define $f = f_L + f_H x^j$ and $g = g_L + g_H x^j$. To obtain $(f_j, g_j)$, it should multiply $M_1$ by origin input $(f, g)$:

$$\begin{pmatrix} f_j \\ g_j \end{pmatrix} = M_1 \begin{pmatrix} f \\ g \end{pmatrix} = M_1 \begin{pmatrix} f_L \\ g_L \end{pmatrix} + M_1 x^j \begin{pmatrix} f_H \\ g_H \end{pmatrix} = \begin{pmatrix} f'_j \\ g'_j \end{pmatrix} + \begin{pmatrix} u_j x & v_j x \\ r_j & s_j \end{pmatrix} \begin{pmatrix} f_H \\ g_H \end{pmatrix}.$$

Next, it jumps another $(n-j)$ division steps from $(\delta_j, f_j, g_j)$ to $(\delta_n, f'_n, g'_n)$ and gets the $(n-j)$-step transition matrix $M_2 = \mathcal{T}_{n-1}...\mathcal{T}_j$. Define $f_j = f_{jL} + f_{jH} x^{n-j}$ and $g_j = g_{jL} + g_{jH} x^{n-j}$. To obtain $(f_n, g_n)$, it should multiply $M_2$ by $(f_j, g_j)$:

$$\begin{pmatrix} f_n \\ g_n \end{pmatrix} = M_2 \begin{pmatrix} f_j \\ g_j \end{pmatrix} = M_2 \begin{pmatrix} f_{jL} \\ g_{jL} \end{pmatrix} + M_2 x^{n-j} \begin{pmatrix} f_{jH} \\ g_{jH} \end{pmatrix} = \begin{pmatrix} f'_n \\ g'_n \end{pmatrix} + \begin{pmatrix} u' x & v' x \\ r' & s' \end{pmatrix} \begin{pmatrix} f_{jH} \\ g_{jH} \end{pmatrix}.$$

Finally, obtain the transition matrix product $P = \mathcal{T}_{n-1}...\mathcal{T}_0 = M_2 M_1$. Algorithm 5 summarizes the operating flow of this scheme. The *jumpdivsteps* function would be called recursively. Replace it with *divstep* while $j$ or $(n-j)$ is small enough such as 4 or 8.

**Algorithm 5** $jumpdivsteps^n(\delta, f, g)$

```
1: $(\delta_j, f'_j, g'_j, u_j, v_j, r_j, s_j) \leftarrow jumpdivsteps^j(\delta, f, g)$
2: $f_j \leftarrow f'_j + (u_j f_H + v_j g_H)x$
3: $g_j \leftarrow g'_j + r_j f_H + s_j g_H$
4: $(\delta_n, f'_n, g'_n, u', v', r', s') \leftarrow jumpdivsteps^{n-j}(\delta_j, f_j, g_j)$
5: $f_n \leftarrow f'_n + (u' f_{jH} + v' g_{jH})x$
6: $g_n \leftarrow g'_n + r' f_{jH} + s' g_{jH}$
7: $u_n \leftarrow u' u_j x + v' r_j$
8: $v_n \leftarrow u' v_j x + v' s_j$
9: $r_n \leftarrow r' u_j x + s' r_j$
10: $s_n \leftarrow r' v_j x + s' s_j$
11: return $(\delta_n, f_n, g_n, u_n, v_n, r_n, s_n)$
```

doi:10.5566/NTU2021XXXXX

# Chapter 3 Implementation

Algorithm 4 provides a common code structure for computing polynomial inversions over different polynomial rings. It takes two parameters: $F$ and $G$. Each polynomial ring in NTRU and Streamlined NTRU Prime has a form: $(Z/q)[x]/P$ where $P$ is a polynomial with degree $d$. To calculate a polynomial inversion over a ring with the above form, set input $F = P$; the other input $G$ could be any element $\in (Z/q)[x]/P$, hence $deg(F) > deg(G)$. From the perspective of implementation, the degree of $G$ could be viewed as $d - 1$, i.e., keeping all terms, including terms with zero coefficient. The final step is to choose an suitable iteration times of the division step: $n \geq deg(F) + deg(G) = 2d - 1$. Table 3.1 shows the parameters used in NTRU and NTRU Prime. Notice that each ring $S_q$ in NTRU has parameter $q \in 2^k$. It takes two steps to calculate polynomial inversion over such rings. The first step is to compute the reciprocal of $G$ over $S_2$. After that, use Hensel lifting method with the result of the first step to calculate the desired answer over $S_q$.

The $jumpdivsteps$ provides an efficient way to speed up the calculation process. It uses several polynomial multiplications; obviously, these operations will affect it the most. The following sections present $jumpdivsteps$ implementation structures designed according to the range of coefficient value—specifically, the size of $q$—and the performance of different polynomial multiplication methods.

doi:10.5566/NTU2021XXXXX

|  | $F$ $(P)$ | $(G, q)$ | $d$ | $n$ |
|---|---|---|---|---|
| **ntruhps2048509** | $\Phi_{509}$ | $(\in S_3, 3)$, $(\in S_q, 2048)$ | 509 | 1024 |
| **ntruhps2048677** | $\Phi_{677}$ | $(\in S_3, 3)$, $(\in S_q, 2048)$ | 677 | 1376 |
| **ntruhps4096821** | $\Phi_{821}$ | $(\in S_3, 3)$, $(\in S_q, 4096)$ | 821 | 1664 |
| **ntruhrss701** | $\Phi_{701}$ | $(\in S_3, 3)$, $(\in S_q, 8192)$ | 701 | 1408 |
| **sntrup653** | $x^{653} - x - 1$ | $(\in R_3, 3)$, $(\in R_q, 4621)$ | 653 | 1312 |
| **sntrup761** | $x^{761} - x - 1$ | $(\in R_3, 3)$, $(\in R_q, 4591)$ | 761 | 1521 |
| **sntrup857** | $x^{857} - x - 1$ | $(\in R_3, 3)$, $(\in R_q, 5167)$ | 857 | 1728 |
| **sntrup953** | $x^{953} - x - 1$ | $(\in R_3, 3)$, $(\in R_q, 6343)$ | 953 | 1920 |
| **sntrup1013** | $x^{1013} - x - 1$ | $(\in R_3, 3)$, $(\in R_q, 7177)$ | 1013 | 2048 |
| **sntrup1277** | $x^{1277} - x - 1$ | $(\in R_3, 3)$, $(\in R_q, 7879)$ | 1277 | 2560 |

Table 3.1: Parameters for polynomial inversion in NTRU and Streamlined NTRU Prime

## 3.1 Jumpdivsteps Algorithm in $(Z/2)[x]/P$

In the ring $(Z/2)[x]/P$, all coefficients of polynomials are in $\{0, 1\}$. A register in ARM can store eight coefficients, i.e., 4 bits for one coefficient. Thus, just one ARM instruction $umull$ or $umlal$ can finish an $8 \times 8$ polynomial multiplication, and the maximum of coefficients of the multiplication result will be 8. Since unsigned 4 bits can only hold values between 0 and 15, the multiplication result may require a reduction before continuing with other operations. A reduction takes only one ARM instruction $and$ as well. Algorithm 6 shows an schoolbook method (long multiplication) to do an $16 \times 16$ polynomial multiplication.

**Algorithm 6** $16 \times 16$ polynomial multiplication (4 bits per coefficient)

```
Input: $f = f_0 + f_1x^8$, $g = g_0 + g_1x^8$
Output: $h = f \times g = h_0 + h_1x^8 + h_2x^{16} + h_3x^{24}$
1: umull $h_0, h_1, f_0, g_0$        $\triangleright (h_0 + h_1x^8) \leftarrow f_0 \times g_0$
2: umull $h_2, h_3, f_1, g_1$        $\triangleright (h_2 + h_3x^8)x^{16} \leftarrow (f_1 \times g_1)x^{16}$
3: umlal $h_1, h_2, f_1, g_0$        $\triangleright (h_1 + h_2x^8)x^8 \leftarrow (f_1 \times g_0)x^8 + (h_1 + h_2x^8)x^8$
4: and $h_1$, #0x11111111
5: and $h_2$, #0x11111111
6: umlal $h_1, h_2, f_0, g_1$        $\triangleright (h_1 + h_2x^8)x^8 \leftarrow (f_0 \times g_1)x^8 + (h_1 + h_2x^8)x^8$
```

Due to the efficiency of the schoolbook polynomial multiplications in the ring $(Z/2)[x]/P$, an variant structure of $jumpdivsteps^n$ are designed to deduct the cost of updating $u$ and

doi:10.5566/NTU2021XXXXX

$r$. This accumulating-base structure presented in Algorithm 7 would first choose an appropriate base size $j$, and it contains $n/j$ phases. A phase consists of two steps: running $jumpdivsteps^j$ and updating $f$, $g$, $v$, and $s$. The $f$, $g$, $v$, and $s$ after executing $k$ phases are as same as the outputs of $jumpdivsteps^{kj}$. Notice that the degrees of $f$ and $g$ will decrease at each phase; on the contrary, the degrees of $v$ and $s$ will increase. Let $deg(f) = deg(g) = d$ where $g(d) = 0$ ($0x^d$). At first, the maximum degree between $f$ and $g$ is $d$. It stays the same until the number of phases reaches $\lfloor d/j \rfloor + 1$. After running $\lfloor d/j \rfloor + 1 + k$ phases, the value will become $d - (k+1)j + d\%j$. On the other hand, the maximum degree between $v$ and $s$ would be $kj$ after running $k$ phases. Hence, the sizes of polynomial multiplications at each phase has a form $j \times (\#current\ maximum\ degree\ of\ input)$.

| | **At Phase** $r$ | |
|---|---|---|
| | $r \leq \lfloor d/j \rfloor$ | $r = \lfloor d/j \rfloor + 1 + k, k \geq 0$ |
| to update $f, g$ | $j \times d$ | $j \times (d - (k+1)j - d\%j)$ |
| to update $v, s$ | $j \times rj$ | |

Table 3.2: Size of polynomial multiplications at each phase

**Algorithm 7** $jumpdivsteps^n(\delta, f, g)$

```
1: $(\delta, u', v', r', s') \leftarrow jumpdivsteps^j(\delta, f, g)$        ▷ phase 0
2: $f \leftarrow (u'f + v'g)x/x^j$        ▷ $j \times d$ polynomial multiplication
3: $g \leftarrow (r'f + s'g)/x^j$        ▷ $j \times d$ polynomial multiplication
4: $v \leftarrow v'$
5: $s \leftarrow s'$
6: $(\delta, u', v', r', s') \leftarrow jumpdivsteps^j(\delta, f, g)$        ▷ phase 1
7: $f \leftarrow (u_j f + v_j g)x/x^j$
8: $g \leftarrow (r_j f + s_j g)/x^j$
9: $v \leftarrow u'vx + v's$        ▷ $j \times j$ polynomial multiplication
10: $s \leftarrow r'vx + s's$        ▷ $j \times j$ polynomial multiplication
11: $(\delta, u', v', r', s') \leftarrow jumpdivsteps^j(\delta, f, g)$        ▷ phase 2
12: $f \leftarrow (u_j f + v_j g)x/x^j$
13: $g \leftarrow (r_j f + s_j g)/x^j$
14: $v \leftarrow u'vx + v's$        ▷ $j \times 2j$ polynomial multiplication
15: $s \leftarrow r'vx + s's$        ▷ $j \times 2j$ polynomial multiplication
16: ... (run $n/j$ times of $jumpdivsteps^j$ in total to update $f, g, v, s$)
17: if $n\%j \neq 0$, then run $jumpdivsteps^{n-kj}$ and update $f, g, v, s$ in the end.
18: return $(\delta, f, v)$
```

doi:10.5566/NTU2021XXXXX

### 3.1.1 Base Structure

The infrastructure development of the accumulating-base structure is mainly composed of $jumpdivsteps^j$ and a code generator of polynomial multiplications.

The suitable value of $j$ could be 32, 64, or 128. There are two ways to implement $jumpdivsteps^{32}$. One is to first implement a $divstep^8$ function; next use the Algorithm 5 to construct $jumpdivsteps^{16}$ and then $jumpdivsteps^{32}$. The other is to develop an $divstep^{32}$ by storing 32 coefficients in a register. Using registers with 32 coefficients for polynomial multiplication is not as efficient as using ones with 8 coefficients. Hence, the second method requires bit conversion for a coefficient at the beginning and end. Despite the additional cost, the second method is still faster than the first one. Core implementation details of $divstep^{32}$ contain three parts: bit conversion, conditional swap, and elimination.

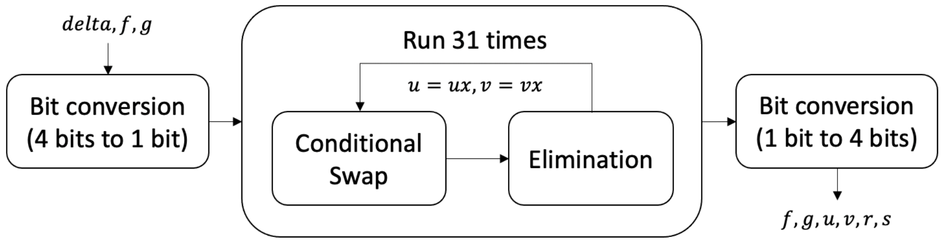

Figure 3.1: Workflow of $divstep^{32}$

In the conditional exchange and elimination part, the register saves the coefficient in reverse. The most significant bit stores the constant term of the polynomial. The $reverse$ feature can be implemented simultaneously within the process of bit conversion. ARM instructions $ubfx$ and $eor$ can complete the bit conversion works. Algorithm 8 is used to process inputs, and Algorithm 9 will be applied to the final outputs. The $IT$ instruction supports up to 4 conditional instructions. These conditional operations satisfy the demand for constant-time computations. Algorithm 10 presents a way to do the conditional swap.

doi:10.5566/NTU2021XXXXX

**Algorithm 8** Bit conversion for 32 coefficients from 4 bits to 1 bit

**Input:** $f = f_0 + f_1x^8 + f_2x^{16} + f_3x^{24}$ (4 bits for one coefficient)
**Output:** $f' = reverse(f)$ (1 bit for one coefficient)

```
1: mov $f', \#0$
2: ubfx $tmp1, f_0, \#0, \#1$
3: ubfx $tmp2, f_0, \#4, \#1$
4: ubfx $tmp3, f_0, \#8, \#1$
5: ubfx $tmp4, f_0, \#12, \#1$
6: eor $f', tmp1, f', LSL\#1$
7: eor $f', tmp2, f', LSL\#1$
8: eor $f', tmp3, f', LSL\#1$
9: eor $f', tmp4, f', LSL\#1$
10: ubfx $tmp1, f_0, \#16, \#1$
11: ubfx $tmp2, f_0, \#20, \#1$
12: ubfx $tmp3, f_0, \#24, \#1$
13: ubfx $tmp4, f_0, \#28, \#1$
14: eor $f', tmp1, f', LSL\#1$
15: eor $f', tmp2, f', LSL\#1$
16: eor $f', tmp3, f', LSL\#1$
17: eor $f', tmp4, f', LSL\#1$                                  ▷ $f' = reverse(f_0)$
18: ... (convert $f_1$, $f_2$, and $f_3$ as the same way as $f_0$)
```

**Algorithm 9** Bit conversion for 32 coefficients from 1 bit to 4 bits

**Input:** $f'$ (4 bits for one coefficient), $f'[i : j] = c_ix^i + ... + c_jx^j$ is a part of $f'$
**Output:** $f = f_0 + f_1x^8 + f_2x^{16} + f_3x^{24} = reverse(f')$ (1 bit for one coefficient)

```
1: ubfx $f_3, f', \#0, \#1$
2: ubfx $tmp, f', \#1, \#1$
3: eor $f_3, tmp, f_3, LSL\#4$
4: ... (same structure for converting bits: #2 to #6)
5: ubfx $tmp, f', \#7, \#1$
6: eor $f_3, tmp, f_3, LSL\#4$                        ▷ $f_3 = reverse(f'[0 : 7])$
7: ubfx $f_2, f', \#8, \#1$
8: ubfx $tmp, f', \#9, \#1$
9: eor $f_2, tmp, f_2, LSL\#4$
10: ... (same structure for converting bits: #10 to #15)   ▷ $f_2 = reverse(f'[8 : 15])$
11: ubfx $f_1, f', \#16, \#1$
12: ubfx $tmp, f', \#17, \#1$
13: eor $f_1, tmp, f_1, LSL\#4$
14: ... (same structure for converting bits: #18 to #23)   ▷ $f_1 = reverse(f'[16 : 23])$
15: ubfx $f_0, f', \#24, \#1$
16: ubfx $tmp, f', \#25, \#1$
17: eor $f_0, tmp, f_0, LSL\#4$
18: ... (same structure for converting bits: #26 to #31)   ▷ $f_0 = reverse(f'[24 : 31])$
```

doi:10.5566/NTU2021XXXXX

**Algorithm 10** Conditional swap in $divstep^{32}$

```
Input: $delta, f, g$
Output: if $delta > 0$ and $g(0) \neq 0$, $swap(f, g)$
1: vcmp.f32 $delta, s1$                  ▷ $delta > 0? (s1 = 1.0)$
2: vmrs $APSR\_nzcv, FPSCR$              ▷ move carry
3: itttt $cs$                            ▷ control by $delta$
4: tstcs $f, g, LSL\#1$                  ▷ set $cs$ by $g(0)$
5: movcs $tmp, f$                        ▷ $if(cs) : swap(f, g)$
6: movcs $f, g$
7: movcs $g, tmp$
```

The elimination part has a computation form in common: $a = f(0)a - g(0)b$. The $f(0)$ is always equal to 1. The truth table drew in Table 3.3 can simplify each bit of $a$, namely $a_i$, to $a_i = a_i \oplus (g(0) \wedge b_i)$. Thus, this equation can be done with two instructions *and* and *eor*.

| $a_i$ | $g(0)$ | $b_i$ | $g(0)b_i$ | $a_i - g(0)b_i$ |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 | 0 |
| 0 | 1 | 0 | 0 | 0 |
| 0 | 1 | 1 | 1 | **1** |
| 1 | 0 | 0 | 0 | **1** |
| 1 | 0 | 1 | 0 | **1** |
| 1 | 1 | 0 | 0 | **1** |
| 1 | 1 | 1 | 1 | 0 |

Table 3.3: Truth table of $a_i - g(0)b_i$

**Algorithm 11** Elimination in $divstep^{32}$

```
Input: $f, g, u, v, r, s$
Output: $g = (g - g(0)f)/x, r = r - g(0)u$, $s = s - g(0)v$
1: and $tmp, u, g, ASR\#31$
2: eors $r, r, tmp$                      ▷ $(r = r - g(0)u)$
3: and $tmp, v, g, ASR\#31$
4: eors $s, s, tmp$                      ▷ $(s = s - g(0)v)$
5: and $tmp, f, g, ASR\#31$
6: eors $g, g, tmp$                      ▷ $(g = g - g(0)f)$
7: lsl $g, g, \#1$                       ▷ $g = g/x$
```

The $jumpdivsteps^{64}$ can be constructed by two $divstep^{32}$ with Algorithm 5. The size of polynomial multiplications in it would be $32 \times 32$. Similarly, The $jumpdivsteps^{128}$ can be constructed by two $jumpdivsteps^{64}$ and $64 \times 64$ polynomial multiplications.

doi:10.5566/NTU2021XXXXX

The process of $32 \times 32$ polynomial multiplications could be decomposed into two shapes: a slim parallelogram and two triangles. Each one is composed of multiple blocks. A block is a result of $8 \times 8$ polynomial multiplication. Due to the maximum value of 4 bits, the addition of two blocks might require one reduction. Let $f = f_0 + f_1 x^8 + f_2 x^{16} + f_3 x^{24}$; $g = g_0 + g_1 x^8 + g_2 x^{16} + g_3 x^{24}$. Left side of Figure 3.2 presents a method to build $f \times g$ by these shapes. The first half process of $32 \times 32$ polynomial multiplication generates the light gray shapes: one slim parallelogram and one inverse triangle; the second half process of it is to produce the dark gray shape: one triangle.

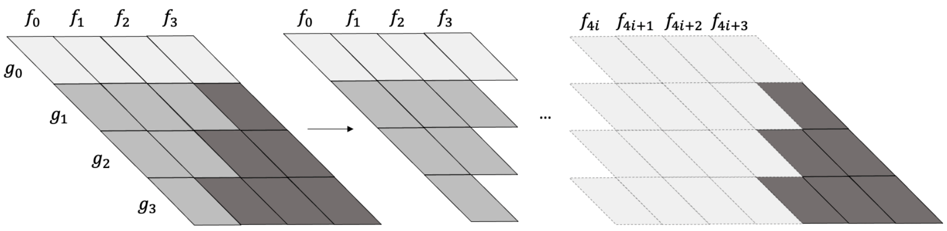

Figure 3.2: Deconstruction of $f \times g$

These shapes can further be the components of the code generator of $32 \times n_2$ polynomial multiplication where $n_2 = 32k$. A $32 \times n_2$ polynomial multiplication could be split into $k$ $32 \times 32$ polynomial multiplications. The dark gray shape of the $k$th $32 \times 32$ polynomial multiplication and the light gray shapes of the $(k+1)$th one form a new shape: big parallelogram. This shape contains four slim parallelograms. Hence, the workflow of the code generator contains three steps. Build the first light gray shapes; generate $k-1$ big parallelograms in the middle; produce the last dark gray shape.

Generalize this structure to $n_1 \times n_2$ where $n_1 = 32i$. Algorithm 12 presents the generation process. $h_{id}$ and $v_{id}$ are used to locate the position of the shapes so that correct coefficients of $f$ and $g$ could be loaded correctly. While $v_{id} = c$, it points to $g_c$ or the $c$th row; $h_{id}$ with the value $c$ points to $f_{4c}$ or $c$th slim parallelogram. Figure 3.3 illustrates the

doi:10.5566/NTU2021XXXXX

concept of them, and it provides a simple example for generating a $64 \times 128$ polynomial multiplication.

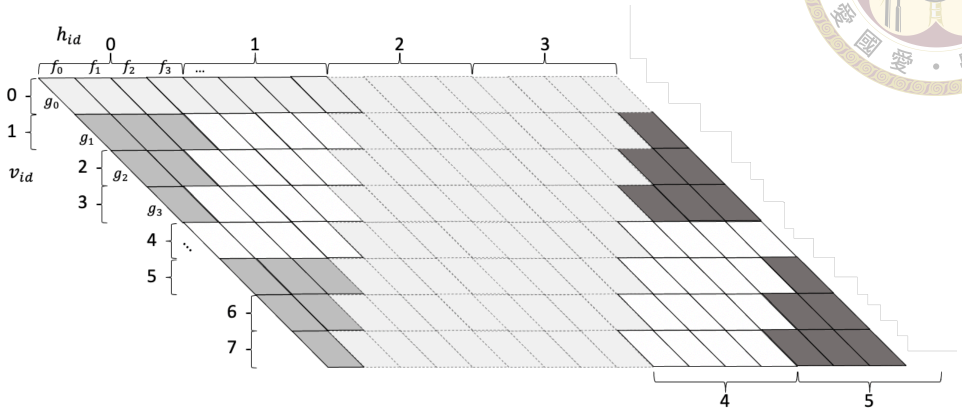

Figure 3.3: Visual graph of $64 \times 128$ polynomial multiplication

**Algorithm 12** Code generator of $n_1 \times n_2$ polynomial multiplication

```
Input: $f, g$
Output: $f \times g$
 1: $i \leftarrow n_1/32$
 2: $k \leftarrow n_2/32$
 3: for $h_{id} \leftarrow 0$ to $i-1$ do                     ▷ The first step:
 4:     generate one slim parallelogram.
 5:     for $v_{id} \leftarrow 1$ to $4i$ do                 ▷ big parallelograms
 6:         generate one slim parallelogram.
 7:     end for
 8:     generate one inverse triangle.                    ▷ light gray shapes
 9: end for
10: for $h_{id} \leftarrow i$ to $k-1$ do                     ▷ The second step:
11:     for $v_{id} \leftarrow 0$ to $4i-1$ do               ▷ big parallelograms
12:         generate one slim parallelogram.
13:     end for
14: end for
15: for $h_{id} \leftarrow k$ to $(i+k)-1$ do                 ▷ The final step:
16:     for $v_{id} \leftarrow i-1$ to $4(h_{id}-k+1)$ do     ▷ big parallelograms
17:         generate one slim parallelogram.
18:     end for
19:     generate one triangle.                            ▷ dark gray shapes
20: end for
21: generate one triangle.                                ▷ dark gray shapes
```

In conclusion, the code generator would first build $i$ light gray shapes plus $i(i-1)/2$ big parallelograms. Next, it generates $i(k-i)$ big parallelograms. Finally, it produces $i$ dark gray shapes plus $i(i-1)/2$ big parallelograms. Algorithm 13 demonstrates a code

doi:10.5566/NTU2021XXXXX

structure to implement light gray shapes. Implementations of other shapes are similar to this one.

**Algorithm 13** Codes for light gray shapes

**Input:** $f, g$

**Output:** $h = h_0 + h_1x^8 + h_2x^{16} + h_3x^{24} = g_0f + (g_1x^8)(f_0 + f_1x^8 + f_2x^{16}) + (g_2x^{16})(f_0 + f_1x^8) + (g_3x^{24})f_0$

```
 1: umull $h_1, h_2, f_1, g_0$                ▷ slim parallelogram
 2: umull $h_3, h_4, f_3, g_0$
 3: umlal $h_0, h_1, f_0, g_0$
 4: umlal $h_2, h_3, f_2, g_0$
 5: and $h_0, h_0$, #0x11111111
 6: and $h_1, h_1$, #0x11111111
 7: and $h_2, h_2$, #0x11111111
 8: and $h_3, h_3$, #0x11111111
 9: and $h_4, h_4$, #0x11111111
10: umlal $h_3, h_4, f_2, g_1$                ▷ inverse triangle
11: umlal $h_2, h_3, f_1, g_1$
12: umlal $h_1, h_2, f_0, g_1$
13: and $h_1, h_1$, #0x11111111
14: and $h_2, h_2$, #0x11111111
15: and $h_3, h_3$, #0x11111111
16: and $h_4, h_4$, #0x11111111
17: umlal $h_3, h_4, f_1, g_2$
18: umlal $h_2, h_3, f_0, g_2$
19: and $h_2, h_2$, #0x11111111
20: and $h_3, h_3$, #0x11111111
21: and $h_4, h_4$, #0x11111111
22: umlal $h_3, h_4, f_0, g_3$
23: and $h_3, h_3$, #0x11111111
24: and $h_4, h_4$, #0x11111111
```

### 3.1.2 Usage in NTRU

There are still a few steps before using Algorithm 7 in NTRU. First, determine the $j$ value for each parameter set of NTRU. The appropriate value of $j$ is related to the $d$, which is the degree of the input $f$. Basically, choosing a large $j$ is a good idea when d is large. However, if $j$ is too large, the performance will slow down. The trade-off could be confirmed through testing. For the parameter sets of NTRU, 64 is a suitable value. Setting

doi:10.5566/NTU2021XXXXX

$j = 128$ could speed up the overall speed little when $d$ is about 830.

Next, generate the multiplications required for each parameter set. The multiplication generator takes 32 as the base unit. However, it can cut the slim parallelogram into half, and it uses the half one to construct a multiplication in units of 16. Therefore, set $d' = \lceil d/16 \rceil d$ and $n = 2d' > 2d - 1$. Finally, adjust the length of $f$ from $d$ to $d'$ by zero-padded coefficients. This constraint also applies to $g$, $v$, and $s$. Table 3.4 lists the spec for polynomial inversion in each parameter set of NTRU.

| | $d$ | $n$ | $d'$ | j | Sizes of polynomial multiplications |
|---|---|---|---|---|---|
| **ntruhps2048509** | 509 | 1024 | 512 | 64 | $64 \times (512 - 64k), 64 \times 544$ |
| **ntruhps2048677** | 677 | 1376 | 688 | 64 | $64 \times 688, 64 \times (672 - 64k), 32 \times 32, 32 \times 704$ |
| **ntruhps4096821** | 821 | 1664 | 832 | 128 | $128 \times (832 - 128k), 128 \times 864$ |
| **ntruhrss701** | 701 | 1408 | 704 | 64 | $64 \times (704 - 64k), 64 \times 736$ |

Table 3.4: Spec of polynomial inversion over $(Z/2)[x]/P$ in NTRU

The final length of the output $v$ may be greater than $d'$. Under the condition: $n - (2d - 1) < 32$, length of $v$ will be less than $(d' + 32)$. As a consequence, just keep up to $d'$ or $(d' + 32)$ coefficients from the multiplication result during updating $v$ in each phase. The same goes for updating $s$.

Table 3.5 shows the performance of getting the polynomial modular reciprocal with Algorithm 4 plus Algorithm 7.

| | Cycles |
|---|---|
| **ntruhps2048509** | 311,283 |
| **ntruhps2048677** | 507,805 |
| **ntruhps4096821** | 699,859 |
| **ntruhrss701** | 526,999 |

Table 3.5: Performance of polynomial inversion over $(Z/2)[x]/P$ in NTRU

doi:10.5566/NTU2021XXXXX

## 3.2 Jumpdivsteps Algorithm in $(Z/3)[x]/P$

The overall code structure of calculating polynomial inversion over the ring $(Z/3)[x]/P$ is the same as the ring $(Z/2)[x]/P$. The difference is that coefficients are stored in 8 bits instead of 4 bits. A register holds four coefficients. Hence, the ARM instructions $umull$ or $umlal$ can complete a $4 \times 4$ polynomial multiplication. After executing the instructions, the maximum value of the coefficients will be 16. The 8-bit space could accommodate the accumulation of $\lfloor 255/16 \rfloor$ polynomial multiplication results.

A full reduction refers to reducing the coefficient value back to a value between 0 to 2. It takes 11 instructions. If reducing a coefficient value to below 30 is acceptable, it takes only 3 instructions. Algorithm 14 presents this method called lazy reduction. An 8-bit number can be expressed as $(a + 16b)$ where $a \leq 15$ and $b \leq 15$. Then, deduce the equation: $a + 16b \equiv a + b\ (mod\ 3) \leq 30$.

**Algorithm 14** $reduction\_lazy(c)$

```
Input: $c$
Output: $c \leq 30$
1: and $b, c$, #0xF0F0F0F0          ▷ $c = a + 16b$
2: and $a, c$, #0x0F0F0F0F
3: add $c, a, b, LSR\#4$            ▷ $c = a + b$
```

When the coefficient value is less than or equal to 5, just two instructions could complete the full reduction.

**Algorithm 15** $reduction\_5(c)$

```
Input: $c \leq 5$
Output: $c \in 0, 1, 2$
1: usub8 $tmp, c$, #0x03030303      ▷ $tmp = c - 3$
2: sel $c, tmp, c$                  ▷ if $tmp \geq 0$, then $c = tmp$
```

When the coefficient value is less than or equal to 11, it requires 5 instructions to

doi:10.5566/NTU2021XXXXX

finish the full reduction. The value could be expressed as $a + 4b$ where $a \leq 3$ and $b \leq 2$. The equation $a + 4b \equiv a + b \ (mod\ 3) \leq 5$ is then derived.

**Algorithm 16** $reduction\_11(c)$

```
Input: $c \leq 11$
Output: $c \in 0, 1, 2$
1: bic $b, c$, #0x03030303            ▷ $c = a + 4b$
2: and $a, c$, #0x03030303
3: add $c, a, b, LSR\#2$              ▷ $c = a + b \leq 5$
4: $reduction\_5(c)$
```

Similarly, when the coefficient value is less than or equal to 32, it takes 8 instructions. The value also could be expressed as $a + 4b$ where $a \leq 3$ and $b \leq 8$. Derive the equation $a + 4b \equiv a + b \ (mod\ 3) \leq 11$. A full reduction for any the input value consists of Lazy reduction and $reduction\_32$.

**Algorithm 17** $reduction\_32(c)$

```
Input: $c \leq 32$
Output: $c \in 0, 1, 2$
1: bic $b, c$, #0x03030303            ▷ $c = a + 4b$
2: and $a, c$, #0x03030303
3: add $c, a, b, LSR\#2$              ▷ $c = a + b \leq 11$
4: $reduction\_11(c)$
```

In summary, lazy reduction is the most cost-effective. Use the lazy reduction instead of the full reduction to maximize performance when the output value is not necessary to be in $\{0, 1, 2\}$. The schoolbook polynomial multiplication remains efficient. The accumulating-base structure to implement $jumpdivsteps^n$ is still suitable.

### 3.2.1 Base Structure

Implementing $divstep^{32}$ over the ring $(Z/3)[x]/P$ is similar to the way over the ring $(Z/2)[x]/P$. The key difference lies in the bit conversion and elimination part.

In $(Z/3)[x]/P$, 32 coefficients can not be held by just one register because the co-

doi:10.5566/NTU2021XXXXX

efficient value might take 2 bits. However, they are held by two registers. The $i$th bits of two registers determine the value of the $i$th coefficient. This technique is called bit-slice. Table 3.6 lists the correspondence between the bit value of the two registers and the coefficient value. $Bit_{i\_X}$ means the $i$th bit of register $X$.

| $(Bit_{i\_A}, Bit_{i\_B})$ | Coefficient value |
|---|---|
| (0, 0) | 0 |
| (1, 0) | 1 |
| (1, 1) | 2 |

Table 3.6: Bit-slice for $(Z/3)[x]$

The bit conversion step has to convert each coefficient from 8 bits to a pair of 1 bit $(Bit_{i\_A}, Bit_{i\_B})$ for inputs. Let $bit_{i\_k}$ be the $k$th bit of the $i$th 8-bit coefficient. $Bit_{i\_A} = bit_{i\_0} \vee bit_{i\_1}$, and $Bit_{i\_B} = bit_{i\_1}$. For outputs, it converts the $(Bit_{i\_A}, Bit_{i\_B})$ back to the $i$th 8-bit coefficient by $Bit_{i\_A} + Bit_{i\_B}$.

| $(Bit_{i\_A}, Bit_{i\_B})$ | 8-bit format |
|---|---|
| (0, 0) | 00000000 |
| (1, 0) | 00000001 |
| (1, 1) | 00000010 |

Table 3.7: Bit conversion between a pair of 1 bit format and 8-bit format

The common equation in the elimination part has to deal with $f(0)$. Rewrite the equation to $a = a - (g(0)/f(0))b$. The $divstep^{32}$ output should be multiplied by $c$ to be the same as the original answer. However, the last step of polynomial inversion in Algorithm 4 will calculate $cv/cf(0) = v/f(0)$. Consequently, this constant multiple can be ignored. Let $d = (g(0)/f(0))b$; $(X_{A_i}, X_{B_i}) = (Bit_{i\_X_A}, Bit_{i\_X_B})$ presents the $i$th coefficient of a polynomial $X$. Simplify $d$ with a truth table showed in Table 3.8. Obtain $d_{A_i} = g_{A_0} \wedge b_{A_i}$; $d_{B_i} = d_{A_i} \wedge (g_{B_0} \oplus f_{B_0} \oplus b_{B_i})$. Finally, get $a = a - d$ in a similar way. $a_{A_i} = (a_{A_i} \oplus d_{A_i}) \vee (a_{B_i} \oplus d_{B_i})$; $a_{B_i} = (a_{B_i} \oplus d_{A_i}) \wedge (d_{B_i} \oplus (a_{A_i} \oplus d_{A_i}))$. Algorithm 18 demonstrates the core codes in the elimination part.

doi:10.5566/NTU2021XXXXX

| $(g_{A_0}, g_{B_0})$ | $(f_{A_0}, f_{B_0})$ | $(b_{A_i}, b_{B_i})$ | $(d_{A_i}, d_{B_i})$ |
|---|---|---|---|
| (1, 1) = 2 | (1, 0) = 1 | (0, 0) = 0 | (0, 0) = 0 |
| (1, 1) = 2 | (1, 0) = 1 | (1, 0) = 1 | (1, 1) = 2 |
| (1, 1) = 2 | (1, 0) = 1 | (1, 1) = 2 | (1, 0) = 1 |
| (0, 0) = 0 | (1, 0) = 1 | (0, 0) = 0 | (0, 0) = 0 |
| (0, 0) = 0 | (1, 0) = 1 | (1, 0) = 1 | (0, 0) = 0 |
| (0, 0) = 0 | (1, 0) = 1 | (1, 1) = 2 | (0, 0) = 0 |
| (1, 0) = 1 | (1, 0) = 1 | (0, 0) = 0 | (0, 0) = 0 |
| (1, 0) = 1 | (1, 0) = 1 | (1, 0) = 1 | (1, 0) = 1 |
| (1, 0) = 1 | (1, 0) = 1 | (1, 1) = 2 | (1, 1) = 2 |
| (1, 1) = 2 | (1, 1) = 2 | (0, 0) = 0 | (0, 0) = 0 |
| (1, 1) = 2 | (1, 1) = 2 | (1, 0) = 1 | (1, 0) = 1 |
| (1, 1) = 2 | (1, 1) = 2 | (1, 1) = 2 | (1, 1) = 2 |
| (0, 0) = 0 | (1, 1) = 2 | (0, 0) = 0 | (0, 0) = 0 |
| (0, 0) = 0 | (1, 1) = 2 | (1, 0) = 1 | (0, 0) = 0 |
| (0, 0) = 0 | (1, 1) = 2 | (1, 1) = 2 | (0, 0) = 0 |
| (1, 0) = 1 | (1, 1) = 2 | (0, 0) = 0 | (0, 0) = 0 |
| (1, 0) = 1 | (1, 1) = 2 | (1, 0) = 1 | (1, 1) = 2 |
| (1, 0) = 1 | (1, 1) = 2 | (1, 1) = 2 | (1, 0) = 1 |

Table 3.8: Truth table of $d = (g(0)/f(0))b$

**Algorithm 18** Elimination in $divstep^{32}$ over the ring $(Z/3)[x]/P$

```
Input: $f, g, a, b$
Output: $a = a - (g(0)/f(0))b$
 1: and $d_A, b_A, g_A, ASR\#31$        ▷ $d_{A_i} = g_{A_0} \wedge b_{A_i}$
 2: eor $d_B, b_B, g_B, ASR\#31$
 3: eors $d_B, d_B, f_B, ASR\#31$
 4: ands $d_B, d_B, d_A$                ▷ $d_{B_i} = d_{A_i} \wedge (g_{B_0} \oplus f_{B_0} \oplus b_{B_i})$
 5: eors $a_A, a_A, d_A$                ▷ $a_{A_i} \oplus d_{A_i}$
 6: eors $d_A, d_A, a_B$                ▷ $a_{B_i} \oplus d_{A_i}$
 7: eors $a_B, a_B, d_B$                ▷ $a_{B_i} \oplus d_{B_i}$
 8: eors $d_B, d_B, a_A$                ▷ $d_{B_i} \oplus (a_{A_i} \oplus d_{A_i})$
 9: orrs $a_A, a_A, a_B$                ▷ $a_{A_i} = (a_{A_i} \oplus d_{A_i}) \vee (a_{B_i} \oplus d_{B_i})$
10: and $a_B, d_A, d_B$                 ▷ $a_{B_i} = (a_{B_i} \oplus d_{A_i}) \wedge (d_{B_i} \oplus (a_{A_i} \oplus d_{A_i}))$
```

On the other hand, the code structure of the multiplication code generator is the same as Algorithm 12 and Algorithm 13 used in the ring $(Z/2)[x]/P$, except for the timing of the reduction. A block is a $4 \times 4$ polynomial multiplication result in the ring $(Z/3)[x]/P$. The Code generator will accumulate blocks to constitute the basic shapes. The accumulation of blocks can be subdivided into two cases to minimize the reduction cost: left half side and right half side.

doi:10.5566/NTU2021XXXXX

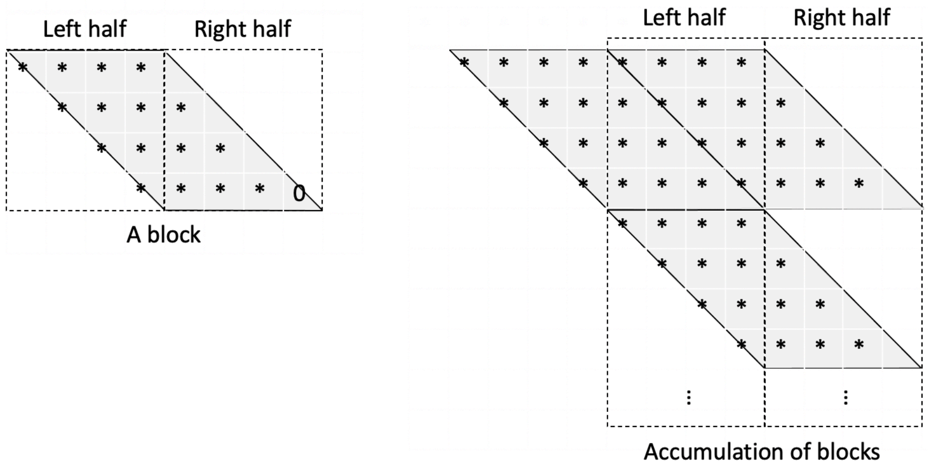

Figure 3.4: Concept of block

The left side of a block contains at most four accumulated coefficients, i.e., the maximum value of coefficients of the block is 16. At most 15 ($\lfloor 255/16 \rfloor$) accumulated blocks are acceptable before the first time lazy reduction. After that, the lazy reduction is required for every 14 ($\lfloor (255-30)/16 \rfloor$) accumulated blocks.

The maximum value of coefficients at the right half of a block is 12. Hence, the first lazy reduction occurs after 21 accumulated blocks. After that, every 18 accumulated blocks requires one lazy reduction.

### 3.2.2 Usage in NTRU and Streamlined NTRU Prime

The appropriate value of $j$ for Algorithm 7 is 128 when $d \geq 761$. Set $j = 64$ when $d \geq 509$. The multiplication code generator takes 16 as the base unit. Therefore, set $d' = \lceil d/16 \rceil d$ and $n = 2d' \geq 2d - 1$. Generate the multiplications required for every parameter sets. Just keep up to $d'$ or $(d' + 16)$ coefficients from the multiplication result during updating $v$ in each phase. The same goes for updating $s$. Table 3.9 lists the spec for polynomial inversion in each parameter set of NTRU; The spec of Streamlined NTRU Prime is in Table 3.10.

doi:10.5566/NTU2021XXXXX

| | $d$ | $n$ | $d'$ | j | Sizes of polynomial multiplications |
|---|---|---|---|---|---|
| **ntruhps2048509** | 509 | 1024 | 512 | 64 | $64 \times (512 - 64k), 64 \times 528$ |
| **ntruhps2048677** | 677 | 1376 | 688 | 64 | $64 \times 688, 64 \times (672 - 64k), 32 \times 32, 32 \times 704$ |
| **ntruhps4096821** | 821 | 1664 | 832 | 128 | $128 \times (832 - 128k), 128 \times 848$ |
| **ntruhrss701** | 701 | 1408 | 704 | 64 | $64 \times (704 - 64k), 64 \times 720$ |

Table 3.9: Spec of polynomial inversion over $(Z/3)[x]/P$ in NTRU

| | $d$ | $n$ | $d'$ | j | Sizes of polynomial multiplications |
|---|---|---|---|---|---|
| **sntrup653** | 653 | 1312 | 656 | 64 | $64 \times 656, 64 \times (608 - 64k),$ $64 \times (64k), 32 \times 32, 32 \times 672$ |
| **sntrup761** | 761 | 1536 | 768 | 128 | $128 \times (768 - 128k), 128 \times 784$ |
| **sntrup857** | 857 | 1728 | 864 | 128 | $128 \times 864, 128 \times (832 - 128k),$ $128 \times (128k), 64 \times 64, 64 \times 880$ |
| **sntrup953** | 953 | 1920 | 960 | 128 | $128 \times 960, 128 \times (896 - 128k), 128 \times 976$ |
| **sntrup1013** | 1013 | 2048 | 1024 | 128 | $128 \times (1024 - 128k), 128 \times 1040$ |
| **sntrup1277** | 1277 | 2560 | 1280 | 128 | $128 \times (1280 - 128k), 128 \times 1296$ |

Table 3.10: Spec of polynomial inversion over $(Z/3)[x]/P$ in Streamlined NTRU Prime

Use Algorithm 4 plus Algorithm 7 to compute the polynomial modular reciprocal. Table 3.11 and Table 3.12 shows the performance in NTRU and Streamlined NTRU Prime respectively.

| | Cycles |
|---|---|
| **ntruhps2048509** | 867,474 |
| **ntruhps2048677** | 1,421,744 |
| **ntruhps4096821** | 1,958,253 |
| **ntruhrss701** | 1,485,067 |

Table 3.11: Performance of polynomial inversion over $(Z/3)[x]/P$ in NTRU

| | Cycles |
|---|---|
| **sntrup653** | 1,211,075 |
| **sntrup761** | 1,576,041 |
| **sntrup857** | 1,897,857 |
| **sntrup953** | 2,265,105 |
| **sntrup1013** | 2,532,837 |
| **sntrup1277** | 3,741,763 |

Table 3.12: Performance of polynomial inversion over $(Z/3)[x]/P$ in Streamlined NTRU Prime

doi:10.5566/NTU2021XXXXX

## 3.3 Jumpdivsteps Algorithm in $(Z/q)[x]/P$

A flaw of Algorithm 7 is the requirement of numerous $n_1 \times n_2$ polynomial multiplications where $n_1 \neq n_2$. Therefore, this structure is not suitable for inefficient polynomial multiplications with different input lengths over the ring $(Z/q)[x]/P$ where $q$ is not a small number. The coefficient value over this ring is between $-q/2$ and $q/2$. A register contains two coefficients, i.e., 16 bits for one coefficient.

### 3.3.1 Base Structure

Use the divide-and-conquer algorithm to implement $jumpdivsteps^n$. It can be first divided into $jumpdivsteps^j$ and $jumpdivsteps^{n-j}$. Let them could be further split into multiple $jumpdivsteps^{2^k}$ subproblems. Algorithm 5 can handle these $jumpdivsteps^{2^k}$ subproblems. Since $v_n$ is the desired output, the calculation of $u_n$, $r_n$, and $s_n$ could be ignored in $jumpdivsteps^n$. Hence, the $jumpdivsteps^j$ and $jumpdivsteps^{n-j}$ only need to compute $(v_j, s_j)$ and $(u', v')$ respectively. Algorithm 19 shows the process.

**Algorithm 19** $jumpdivsteps^n(\delta, f, g)$

```
1: $(\delta_j, f'_j, g'_j, v_j, s_j) \leftarrow jumpdivsteps^j(\delta, f, g)$
2: $f_j \leftarrow f'_j + (u_j f_H + v_j g_H)x$
3: $g_j \leftarrow g'_j + r_j f_H + s_j g_H$
4: $(\delta_n, f'_n, g'_n, u', v') \leftarrow jumpdivsteps^{n-j}(\delta_j, f_j, g_j)$
5: $f_n \leftarrow f'_n + (u' f_{jH} + v' g_{jH})x$
6: $g_n \leftarrow g'_n + r' f_{jH} + s' g_{jH}$
7: $v_n \leftarrow u' v_j x + v' s_j$
8: return $(\delta_n, f_n, g_n, v_n)$
```

The size of polynomial multiplications in $jumpdivsteps^{2^k}$ would all be $2^{k-1} \times 2^{k-1}$. There are 16 in total. These multiplications could be further optimized by Karatsuba, Toom, or NTT. On the other hand, the size of polynomial multiplications in $jumpdivsteps^j$

doi:10.5566/NTU2021XXXXX

and $jumpdivsteps^{n-j}$ might be $2^k \times 2^{k'}$ where $k \leq k'$ or $c \times 2^{k'}$. Such multiplications could still be composed of multiple groups of $2^k \times 2^k$ multiplications which are implemented by Karatsuba or Toom. However, the performance of $c \times 2^{k'}$ might not be good. NTT method might be a good choice for $c \times 2^{k'}$.

The NTT method has an advantage in Algorithm 5. The 16 ordinary multiplications would become pointwise multiplications which takes time $\Theta(n)$. The conversion cost between representation forms would drastically reduce due to the reusability of the point-value of $u$, $v$, $r$, and $s$. Although the NTT method might be slower than other methods at the single polynomial multiplication test, it would be the fastest at the overall performance test. There are five main functions in NTT implementation: $ntt$, $ntt\_x$, $intt$, $intt\_x$, and $basemul$. Both $ntt$ and $ntt\_x$ take a polynomial in the coefficient form as input. The former would return the polynomial in the point-value form. The latter would return the product of the polynomial and $x$ in the point-value form. Reversely, the $intt$ returns the input polynomial in the coefficient form. The $intt\_x$ returns the division of the input polynomial by $x$ in the coefficient form. The $basemul$ returns the pointwise multiplication of two inputs. Algorithm 20 presents the NTT implementation of Algorithm 5.

### 3.3.2 Usage in Streamlined NTRU Prime

Each parameter sets of Streamlined NTRU Prime has $n \geq 2d - 1$. Extend the length of inputs $f$ and $g$ of $jumpdivsteps^n$ to $d'$ with zero-padded coefficients. It satisfies $d' = j \geq d$ and $(n - j) \geq (2d - 1) - d'$. Hence, $jumpdivsteps^n$ could be simplified to Algorithm 21. $jumpdivsteps^{n-j}$ has a special condition that the input length is equal to $n - j$. The code structure of it is similar to Algorithm 5. However, the size of polynomial multiplication for updating $f$ and $g$ could be $(c_{length} - 2^k) \times 2^k$ instead of $c_{length} \times 2^k$

doi:10.5566/NTU2021XXXXX

**Algorithm 20** NTT implementation of $jumpdivsteps^n(\delta, f, g)$

```
1: $(\delta_j, f'_j, g'_j, u_j, v_j, r_j, s_j) \leftarrow jumpdivsteps^j(\delta, f, g)$
2: $Ux \leftarrow ntt\_x(u_j)$
3: $Vx \leftarrow ntt\_x(v_j)$
4: $R \leftarrow ntt(r_j)$
5: $S \leftarrow ntt(s_j)$
6: $F_H \leftarrow ntt(f_H)$
7: $G_H \leftarrow ntt(g_H)$
8: $f_j \leftarrow f'_j + intt(basemul(Ux, F_H) + basemul(Vx, G_H))$
9: $g_j \leftarrow g'_j + intt(basemul(R, F_H) + basemul(S, G_H))$
10: $(\delta_n, f'_n, g'_n, u', v', r', s') \leftarrow jumpdivsteps^{n-j}(\delta_j, f_j, g_j)$
11: $U'x \leftarrow ntt\_x(u')$
12: $V'x \leftarrow ntt\_x(v')$
13: $R' \leftarrow ntt(r')$
14: $S' \leftarrow ntt(s')$
15: $F'_H \leftarrow ntt(f_{jH})$
16: $G'_H \leftarrow ntt(g_{jH})$
17: $f_n \leftarrow f'_n + intt(basemul(U'x, F'_H) + basemul(V'x, G'_H))$
18: $g_n \leftarrow g'_n + intt(basemul(R', F'_H) + basemul(S', G'_H)$
19: $u_n \leftarrow intt\_x(basemul(U'x, Ux) + basemul(V'x, R))$
20: $v_n \leftarrow intt\_x(basemul(U'x, Vx) + basemul(V'x, S))$
21: $r_n \leftarrow intt(basemul(R', Ux) + basemul(S', R))$
22: $s_n \leftarrow intt(basemul(R', Vx) + basemul(S', S))$
23: return $(\delta_n, f_n, g_n, u_n, v_n, r_n, s_n)$
```

where $c_{length}$ is the number of coefficients of current $f$ and $g$.

**Algorithm 21** $jumpdivsteps^n(\delta, f, g)$

```
1: $(\delta_j, f_j, g_j, v_j, s_j) \leftarrow jumpdivsteps^j(\delta, f, g)$
2: $(\delta_n, f_n, g_n, u', v') \leftarrow jumpdivsteps^{n-j}(\delta_j, f_j, g_j)$
3: $v_n \leftarrow u'v_jx + v's_j$
4: return $(\delta_n, f_n, g_n, v_n)$
```

The following introduces the $jumpdivstep^n$ structure used in each parameter sets. The polynomial multiplications would be implemented by three methods: Toom, NTT-Rader, and NTT-big prime. The current $sntrup761$ in $pqm4$ project uses Karatsuba and Toom method. The Toom code generator is designed for $sntrup761$. It can not handle multiplications used in other parameter sets with a larger $q$. This thesis adjusts the timing point of the reduction work in the code generator to support multiplications in other parameter sets. Analogous to the thesis of Cheng [9], the NTT-Rader polynomial mul-

doi:10.5566/NTU2021XXXXX

tiplications can be produced by adjusting the size of NTTs provided in that thesis. The NTT-big prime method mentioned in that thesis provides a size $2^n$ NTT way to optimize multiplications. This thesis provides the implementations of 256, 512, and 1024 NTT.

| Size | 16 bits | | | 32 bits | | | |
|---|---|---|---|---|---|---|---|
| | **ntt** | **intt** | **basemul/basemul_x** | **ntt** | **intt** | **basemul/basemul_x** | **crt** |
| 256 | 4,569 | 5,395 | 2,387/1,647 | 6,190 | 6664 | 3123/1,456 | 6,622 |
| 512 | 8,889 | 10,519 | 5,941/3,088 | 12,108 | 12,873 | 9,176/2,512 | 13,151 |
| 1024 | 24,198 | 28,388 | 9,299/6,352 | 32,813 | 34,641 | 12,248/5,712 | 26,202 |

Table 3.13: Performance of NTT-big prime

#### 3.3.2.1 sntrup653

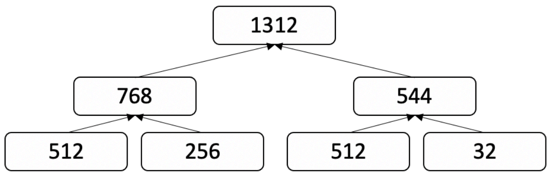

Figure 3.5: Deconstruction of $jumpdivsteps^{1312}$ for sntrup653

Set $(n, d') = (1312, 768)$. Figure 3.5 shows the divide-and-conquer structure of $jumpdivsteps^{1312}$. Table 3.14 summarizes the required multiplications and the performance of them implemented by different methods.

| **Size** | **Toom** | **NTT-Rader** | **NTT-big prime** |
|---|---|---|---|
| $128 \times 128$ | 15,210 | 24,808 | 43,772 |
| $256 \times 256$ | 44,977 | 49,337 | 90,436 |
| $32 \times 512$ | - | 74,296 | - |
| $256 \times 512$ | - | 74,296 | - |
| $653 \times 653$ | 208,592 | 112,243 | - |

Table 3.14: Performance of polynomial multiplications in **sntrup653**

The NTT-Rader method is the fastest except for the size of $128 \times 128$. Use the fastest multiplications to implement each $jumpdivsteps$. Table 3.15 shows the performance of them.

doi:10.5566/NTU2021XXXXX

|  | Polynomial multiplications | Cycles |
|---|---|---|
| **jump256divsteps** | $128 \times 128$ (NTT-Rader) | 596,138 |
| **jump512divsteps** | $256 \times 256$ (NTT-Rader) | 1,617,286 |
| **jump768divsteps** | $256 \times 512$ (NTT-Rader) | 2,761,254 |
| **jump544divsteps** | $32 \times 32$ (NTT-Rader), $32 \times 512$ (NTT-Rader) | 1,922,752 |
| **jump1312divsteps** | $653 \times 653$ (NTT-Rader) | 4,943,427 |

Table 3.15: Performance of $jumpdivsteps$ in **sntrup653**

#### 3.3.2.2 sntrup761

This thesis keeps the original structure of $jumpdivsteps^{1521}$ in $pqm4$ and focuses on the optimization of multiplications. Table 3.16 summarizes the required multiplications and the performance of them implemented by different methods.

| **Size** | **Toom** | **NTT-Rader** | **NTT-big prime** |
|---|---|---|---|
| $128 \times 128$ | 15,477 | 26,415 | 43,772 |
| $256 \times 256$ | 44,977 | 53,461 | 90,436 |
| $256 \times 512$ | - | 83,828 | - |
| $768 \times 768$ | - | 112,243 | - |

Table 3.16: Performance of polynomial multiplications in **sntrup761**

The Toom method is the fastest in each size. However, the NTT-Rader method can utilize the code structure introduced in Algorithm 20 to reduce the overall cost. In terms of the overall performance of $jumpdivsteps$, it does perform better than the Toom method through testing. Table 3.17 presents the final choices for multiplications and the overall performance of each $jumpdivsteps$.

|  | Polynomial multiplications | Cycles |
|---|---|---|
| **jump256divsteps** | $128 \times 128$ (NTT-Rader) | 611,359 |
| **jump512divsteps** | $256 \times 256$ (NTT-Rader) | 1,686,895 |
| **jump768divsteps** | $512 \times 512$ (NTT-Rader) | 2,916,146 |
| **jump753divsteps** | $16 \times 16$, $32 \times 32$, $64 \times 64$ (Karatsuba), $128 \times 128$ (NTT-Rader), $256 \times 512$ (NTT-Rader) | 2,609,118 |
| **jump1536divsteps** | $768 \times 768$ (NTT-Rader) | 5,819,953 |

Table 3.17: Performance of $jumpdivsteps$ in **sntrup761**

doi:10.5566/NTU2021XXXXX

#### 3.3.2.3 sntrup857

Set $(n, d') = (1728, 1024)$. Figure 3.6 shows the divide-and-conquer structure of $jumpdivsteps^{1728}$. Table 3.18 summarizes the required multiplications and the performance of them implemented by different methods.

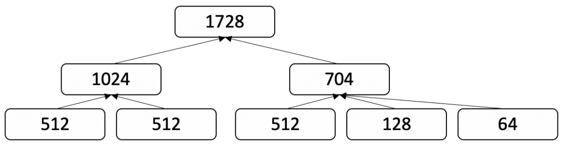

Figure 3.6: Deconstruction of $jumpdivsteps^{1728}$ for sntrup857

| Size | Toom | NTT-Rader | NTT-big prime |
|---|---|---|---|
| $128 \times 128$ | 15,210 (Karatsuba) | - | 43,772 |
| $192 \times 192$ | 28,441 | - | - |
| $256 \times 256$ | 44,977 | 64,866 | 90,436 |
| $192 \times 512$ | 100,662 | - | - |
| $512 \times 512$ | 131,347 | 83,828 | 216,378 |
| $857 \times 857$ | 299,638 | 112,243 | - |

Table 3.18: Performance of polynomial multiplications in **sntrup857**

The NTT-Rader method is the fastest when the size of polynomials is beyond 512. Although it is slower than the Toom method when the size of polynomials is equal to 256, it still performs better than that in the aspect of the overall performance of $jumpdivsteps$. When the size of polynomials is less than 256, the toom method is faster than the NTT-big prime method, including at the test of overall performance.

| | Polynomial multiplications | Cycles |
|---|---|---|
| **jump256divsteps** | $128 \times 128$ (Karatsuba) | 650,990 |
| **jump512divsteps** | $256 \times 256$ (NTT-Rader) | 1,813,422 |
| **jump1024divsteps** | $512 \times 512$ (NTT-Rader) | 4,541,674 |
| **jump704divsteps** | $64 \times 64$ (Karatsuba), $64 \times 128$ (Karatsuba), $192 \times 192$ (Toom), $192 \times 512$ (Toom) | 2,661,931 |
| **jump1728divsteps** | $857 \times 857$ (NTT-Rader) | 7,614,216 |

Table 3.19: Performance of $jumpdivsteps$ in **sntrup857**

doi:10.5566/NTU2021XXXXX

#### 3.3.2.4 sntrup953

Set $(n, d') = (1920, 1024)$. Figure 3.7 shows the divide-and-conquer structure of $jumpdivsteps^{1920}$. Table 3.20 summarizes the required multiplications and the performance of them implemented by different methods.

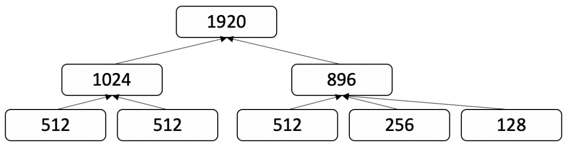

Figure 3.7: Deconstruction of $jumpdivsteps^{1920}$ for sntrup953

| Size | Toom | NTT-Rader | NTT-big prime |
|---|---|---|---|
| $128 \times 128$ | 15,210 (Karatsuba) | - | 43,772 |
| $128 \times 256$ | 33,268 (Karatsuba) | - | - |
| $256 \times 256$ | 44,977 | - | 90,436 |
| $384 \times 384$ | 82,559 | - | - |
| $384 \times 512$ | 136,528 | - | - |
| $512 \times 512$ | 131,347 | 119,420 | 216,378 |
| $953 \times 953$ | 332,606 | 243,394 | 507,290 |

Table 3.20: Performance of polynomial multiplications in **sntrup953**

The final choices of multiplication methods are similar to $sntrup857$. When the size is beyond 512, use the NTT-Rader method instead of the Toom method. Table 3.21 presents the final choices for multiplications and the overall performance of each $jumpdivsteps$.

| | Polynomial multiplications | Cycles |
|---|---|---|
| **jump256divsteps** | $128 \times 128$ (Karatsuba) | 650,990 |
| **jump512divsteps** | $256 \times 256$ (Toom) | 2,054,584 |
| **jump1024divsteps** | $512 \times 512$ (NTT-Rader) | 5,202,521 |
| **jump896divsteps** | $128 \times 128$ (Karatsuba), $128 \times 256$ (Karatsuba), $384 \times 384$ (Toom), $512 \times 512$ (NTT-Rader) | 3,933,811 |
| **jump1920divsteps** | $953 \times 953$ (NTT-Rader) | 9,652,579 |

Table 3.21: Performance of $jumpdivsteps$ in **sntrup953**

doi:10.5566/NTU2021XXXXX

#### 3.3.2.5 sntrup1013

Set $(n, d') = (2048, 1024)$. Figure 3.8 shows the divide-and-conquer structure of $jumpdivsteps^{2048}$. Notice that one of the $jumpdivsteps^{1024}$ will only output $v$ and $s$. The other will only output $u$, $v$. Table 3.20 summarizes the required multiplications and the performance of them implemented by different methods. They are denoted as $jump1024divsteps\_only\_vs$ and $jump1024divsteps\_only\_uv$ respectively.

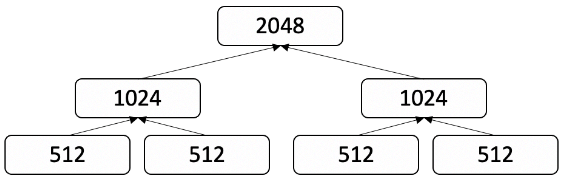

Figure 3.8: Deconstruction of $jumpdivsteps^{2048}$ for sntrup1013

| Size | Toom | NTT-Rader | NTT-big prime |
|---|---|---|---|
| $128 \times 128$ | 15,210 (Karatsuba) | 31,812 | 43,772 |
| $256 \times 256$ | 45,233 | 63,318 | 90,436 |
| $512 \times 512$ | 134,832 | 128,224 | 216,378 |
| $1024 \times 1024$ | 368,922 | 282,633 | 507,290 |

Table 3.22: Performance of polynomial multiplications in **sntrup1013**

The advantages of NTT in Algorithm 20 will not be able to use when $size \geq 512$. The RAM in Cortex-M4 is not enough to store all point-value of $u$, $v$, $r$, and $s$. However, the NTT-Rader method is faster than the Toom one in the single polynomial multiplication test when the size is greater than or equal to 512. This method still should be selected. In conclusion, use the NTT-Rader method instead of the Toom method when $size \geq 256$.

| | Polynomial multiplications | Cycles |
|---|---|---|
| **jump256divsteps** | $128 \times 128$ (Karatsuba) | 649,878 |
| **jump512divsteps** | $256 \times 256$ (NTT-Rader) | 1,828,535 |
| **jump1024divsteps_only_vs** | $512 \times 512$ (NTT-Rader) | 5,205,203 |
| **jump1024divsteps_only_uv** | $512 \times 512$ (NTT-Rader) | 4,700,380 |
| **jump2048divsteps** | $1024 \times 1024$ (NTT-Rader) | 10,543,724 |

Table 3.23: Performance of $jumpdivsteps$ in **sntrup1024**

doi:10.5566/NTU2021XXXXX

#### 3.3.2.6 sntrup1277

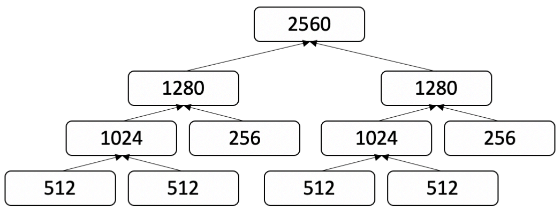

Figure 3.9: Deconstruction of $jumpdivsteps^{2560}$ for sntrup1277

Set $(n, d') = (2560, 1280)$. Figure 3.9 shows the divide-and-conquer structure of $jumpdivsteps^{2560}$. Notice that one of the $jumpdivsteps^{1280}$ will only output $v$ and $s$. The other will only output $u$, $v$. They are denoted as $jump1280divsteps\_only\_vs$ and $jump1280divsteps\_only\_uv$ respectively. Table 3.24 summarizes the required multiplications and the performance of them implemented by different methods.

| **Size** | **Toom** | **NTT-Rader** | **NTT-big prime** |
|---|---|---|---|
| $128 \times 128$ | 15,210 (Karatsuba) | - | 43,772 |
| $256 \times 256$ | 45,233 | - | 90,436 |
| $512 \times 512$ | 131,859 | - | 216,378 |
| $256 \times 1024$ | 191,129 | - | - |
| $1280 \times 1280$ | 510,824 | 342,398 | 558,347 |

Table 3.24: Performance of polynomial multiplications in **sntrup1277**

There has the same space problem for the NTT method similar to $sntrup1013$. Hence, use the Toom method except for the $1280 \times 1280$ polynomial multiplication.

| | Polynomial multiplications | Cycles |
|---|---|---|
| **jump256divsteps** | $128 \times 128$ (Karatsuba) | 651,595 |
| **jump512divsteps** | $256 \times 256$ (Toom) | 2,060,132 |
| **jump1024divsteps** | $512 \times 512$ (Toom) | 6,297,652 |
| **jump1280divsteps_only_vs** | $256 \times 1024$ (Toom) | 9,301,694 |
| **jump1280divsteps_only_uv** | $256 \times 256$ (Toom), $256 \times 1024$ (Toom) | 7,917,391 |
| **jump2560divsteps** | $1280 \times 1280$ (NTT-Rader) | 17,951,693 |

Table 3.25: Performance of $jumpdivsteps$ in **sntrup1277**

doi:10.5566/NTU2021XXXXX

# Chapter 4 Results

The previous works are the current state-of-the-art implementations in the $pqm4$ project. The code of this work has been merged into this project.

## 4.1 NTRU

| | Speed (cycles) | | |
|---|---|---|---|
| | Key generation | Encapsulation | Decapsulation |
| ntruhps2048509 | 79,237,178 | 644,070 | 533,860 |
| ntruhps2048677 | 136,547,633 | 938,031 | 809,883 |
| ntruhps4096821 | 210,327,784 | 1,176,089 | 1,024,692 |
| ntruhrss701 | 153,139,384 | 381,208 | 863,032 |

Table 4.1: Performance of NTRU previous work

| | Speed (cycles) | | |
|---|---|---|---|
| | Key generation | Encapsulation | Decapsulation |
| ntruhps2048509 | 2,940,581 | 644,071 | 533,857 |
| ntruhps2048677 | 4,740,861 | 938,031 | 809,883 |
| ntruhps4096821 | 6,262,187 | 1,176,090 | 1,024,691 |
| ntruhrss701 | 4,235,536 | 381,207 | 863,031 |

Table 4.2: Performance of NTRU

## 4.2 Streamlined NTRU Prime

doi:10.5566/NTU2021XXXXX

|  | Speed (cycles) |  |  |
|---|---|---|---|
|  | **Key generation** | **Encapsulation** | **Decapsulation** |
| sntrup653 | 96,170,999 | 633,156 | 487,025 |
| sntrup761 | 10,852,491 | 684,336 | 537,940 |
| sntrup857 | 169,066,931 | 853,555 | 687,965 |
| sntrup953 | 210,716,799 | 943,842 | 739,499 |
| sntrup1013 | 229,290,578 | 1,033,009 | 838,473 |
| sntrup1277 | 362,563,721 | 1,325,943 | 1,070,579 |

Table 4.3: Performance of Streamlined NTRU Prime previous work

|  | Speed (cycles) |  |  |
|---|---|---|---|
|  | **Key generation** | **Encapsulation** | **Decapsulation** |
| sntrup653 | 6,650,815 | 629,265 | 486,847 |
| sntrup761 | 7,949,876 | 682,008 | 534,547 |
| sntrup857 | 10,257,849 | 852,057 | 685,000 |
| sntrup953 | 12,744,797 | 941,634 | 741,212 |
| sntrup1013 | 13,983,083 | 1,030,507 | 833,363 |
| sntrup1277 | 22,853,961 | 1,320,360 | 1,066,752 |

Table 4.4: Performance of Streamlined NTRU Prime

doi:10.5566/NTU2021XXXXX

# Chapter 5 Conclusion and Future Works

The core cost of polynomial inversion lies in $jumpdivsteps^n$. The $jumpdivsteps^n$ is broken down into multiple subproblems. Each subproblem uses many polynomial multiplications. Therefore, effective polynomial multiplication can greatly speed up the overall speed. Generally speaking, large inputs will be implemented by fast multiplication algorithms. However, ARM provides $umull$ and $umlal$ instructions, making the schoolbook method extremely efficient over the rings $(Z/2)[x]/P$ and $(Z/3)[x]/P$. In addition, when the coefficient is small enough, the bitwise operation can process 32 coefficients at one time, offering a great performance of $divstep^{32}$. The way to deconstruct $jumpdivsteps^n$ will affect the size of polynomial multiplication. This work uses multiple $2^k$ size subproblems over the ring $(Z/q)[x]/P$. The size is good for fast multiplication algorithms to implement multiplication. The NTT method has great performance when the multiplication size is greater than or equal to 512.

Other ARM processors with the instruction set of Cortex-M4 could reuse this work. The implementation of polynomial inversions over $(Z/2)[x]/P$ and $(Z/3)[x]/P$ could be painlessly applied to other cryptosystems. The implementation of polynomial inversions over $(Z/q)[x]/P$ uses a lot of efficient multiply instructions provided by Cortex-M4, such

doi:10.5566/NTU2021XXXXX

as $smlabb$, $smuad$, $smlad$, etc. If other processors can not provide effective instructions with the same function, the performance may be greatly reduced. However, the multiplication code generators used in ring $(Z/2)[x]/P$ and $(Z/3)[x]/P$ have high compatibility with other processors. They only use basic multiply instructions: $umull$ and $umlal$. They should be able to maintain the same performance on other processors.

doi:10.5566/NTU2021XXXXX

# References

[1] M. J. Kannwischer, J. Rijneveld, P. Schwabe, and K. Stoffelen, "Pqm4: Testing and benchmarking NIST PQC on ARM Cortex-M4," IACR Cryptol. ePrint Arch., vol. 2019, p. 844, 2019.

[2] M. J. Kannwischer, J. Rijneveld, and P. Schwabe, "Faster multiplication in $\mathbb{Z}_{2^m}[x]$ on Cortex-M4 to speed up NIST PQC candidates," in ACNS, 2019.

[3] 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: Comparison of optimization strategies on Cortex-M4," IACR Transactions on Cryptographic Hardware and Embedded Systems, vol. 2021, no. 1, pp. 217–238, Dec. 2020. DOI: 10.46586/tches.v2021.i1.217-238. [Online]. Available: https://tches.iacr.org/index.php/TCHES/article/view/8733.

[4] D. Bernstein, C. Chuengsatiansup, T. Lange, and C. V. Vredendaal, "NTRU Prime: Reducing attack surface at low cost," in SAC, 2017.

[5] D. J. Bernstein and B.-Y. Yang, "Fast constant-time gcd computation and modular inversion," IACR Transactions on Cryptographic Hardware and Embedded Systems, vol. 2019, no. 3, pp. 340–398, May 2019. DOI: 10.13154/tches.v2019.i3.340-398. [Online]. Available: https://tches.iacr.org/index.php/TCHES/article/view/8298.

[6] J. Hoffstein, J. Pipher, and J. H. Silverman, "NTRU: A new high speed public key cryptosystem," draft from at CRYPTO '96 rump session, 1996, [Online]. Available: https://ntru.org/f/hps96.pdf.

doi:10.5566/NTU2021XXXXX

[7] C. Chen, O. Danba, and J. Hoffstein. (Mar. 30, 2019). "NTRU algorithm specifications and supporting documentation," [Online]. Available: https://ntru.org/f/ntru-20190330.pdf.

[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. (Oct. 7, 2020). "NTRU Prime: Round 3," [Online]. Available: https://ntruprime.cr.yp.to/nist/ntruprime-20201007.pdf.

[9] Y.-L. Cheng, "Number theoretic transform for polynomial multiplication in lattice-based cryptography on ARM processors," 2021.

doi:10.5566/NTU2021XXXXX