Western Graduate&PostdoctoralStudies

Western University
Scholarship@Western

Electronic Thesis and Dissertation Repository

4-24-2018 11:30 AM

Putting Fürer's Algorithm into Practice with the BPAS Library

Linxiao Wang
The University of Western Ontario

Supervisor
Moreno Maza, Marc
The University of Western Ontario

Graduate Program in Computer Science
A thesis submitted in partial fulfillment of the requirements for the degree in Master of Science
© Linxiao Wang 2018

Follow this and additional works at: https://ir.lib.uwo.ca/etd

 Part of the Algebra Commons, and the Other Computer Sciences Commons

Recommended Citation

Wang, Linxiao, "Putting Fürer's Algorithm into Practice with the BPAS Library" (2018). *Electronic Thesis and Dissertation Repository*. 5358.
https://ir.lib.uwo.ca/etd/5358

This Dissertation/Thesis is brought to you for free and open access by Scholarship@Western. It has been accepted for inclusion in Electronic Thesis and Dissertation Repository by an authorized administrator of Scholarship@Western. For more information, please contact wlswadmin@uwo.ca.

Abstract

Fast algorithms for integer and polynomial multiplication play an important role in scientific computing as well as other disciplines. In 1971, Schönhage and Strassen designed an algorithm that improved the multiplication time for two integers of at most n bits to $O(\log n \log\log n)$. In 2007, Martin Fürer presented a new algorithm that runs in $O\left(n \log n \cdot 2^{O(\log^* n)}\right)$, where $\log^* n$ is the iterated logarithm of n.

We explain how we can put Fürer's ideas into practice for multiplying polynomials over a prime field $\mathbb{Z}/p\mathbb{Z}$, which characteristic is a Generalized Fermat prime of the form $p = r^k + 1$ where k is a power of 2 and r is of machine word size. When k is at least 8, we show that multiplication inside such a prime field can be efficiently implemented via Fast Fourier Transform (FFT). Taking advantage of Cooley-Tukey tensor formula and the fact that r is a $2k$-th primitive root of unity, we obtain an efficient implementation of FFT over $\mathbb{Z}/p\mathbb{Z}$. This implementation outperforms comparable implementations either using other encodings of $\mathbb{Z}/p\mathbb{Z}$ or other ways to perform multiplication in $\mathbb{Z}/p\mathbb{Z}$.

Keywords: Fürer's algorithm, Fast Fourier Transform, Generalized Fermat prime, polynomial multiplication

Contents

List of Algorithms

List of Source Code Listing

List of Figures

List of Tables

List of Appendices

Chapter 1

Introduction

Asymptotically fast algorithms for exact polynomial and matrix arithmetic play a central role in scientific computing. Among others, the discoveries of Karatsuba [21], Cooley and Tukey [6], Strassen [27], and Schönhage and Strassen [25] have initiated an intense activity in both numerical computing and symbolic computation. The implementation of asymptotically fast algorithms is, on its own, a research direction. Often the theoretical analysis of asymptotically fast algorithms focuses on arithmetic operation counts, thus ignoring important hardware details, in particular costs of memory accesses. On modern hardware architectures, these theoretical simplifications are questionable and other complexity measures, such as cache complexity [14], are needed to better analyze algorithms.

In the past two decades, several software for performing symbolic computations have put a great deal of effort in providing outstanding performance, including successful implementation of asymptotically fast arithmetic. As a result, the general-purpose computer algebra system Magma [5] and the Number Theory Library NTL [26] have set world records for polynomial factorization and determining orders of elliptic curves. The book *Modern Computer Algebra* [17] has also contributed to increase the general interest of the computer algebra community for these algorithms. As for linear algebra, in addition to Magma, let us mention the C++ template library LinBox [20] for exact, high-performance linear algebra computation with dense, sparse, and structured matrices over the integers and over finite fields. A cornerstone of this library is the use of BLAS libraries such as ATLAS to provide high-speed routines for matrices over small finite fields, through floating-point computations [11].

The algorithm of Schönhage and Strassen [25] is an asymptotically fast algorithm for multiplying integers in arbitrary precision. It uses the fast Fourier transform (FFT) and, for two integers of at most n bits, it computes their product in $O(n \log n \cdot \log\log n)$ bit operations. This result remained the best known upper bound until the celebrated paper of Martin Fürer [15]. His integer multiplication algorithm runs in $O\left(n \log n \cdot 2^{O(\log^* n)}\right)$ bit operations, where $\log^* n$ is the iterated logarithm of n, defined as:

$$\log^* n := \begin{cases} 0 & \text{if } n \leq 1; \\ 1 + \log^*(\log n) & \text{if } n > 1 \end{cases} \tag{1.1}$$

A detailed analysis suggests that Fürer's algorithm is expected to outperform that of Schönhage and Strassen for $n \geq 2^{2^{64}}$.

The practicality of Fürer's algorithm is an open question. And, in fact, the work reported in this thesis is a contribution responding to this open question. Before presenting our approach, we observe that the ideas of Fürer are not specific to integer multiplication and can be used for multiplying polynomials with coefficients in the field of complex numbers or in a finite field. In [9, 10] Anindya De, Piyush P. Kurur, Chandan Saha and Ramprasad Saptharishi gave a similar algorithm which relies on *finite field arithmetic* and achieves the same running time as Fürer's algorithm. Working with polynomials with coefficients in a finite field will be our framework. Hereafter, we explain the "main trick" of Fürer's algorithm. To this end, we follow an analysis reported by Liangyu Chen, Svyatoslav Covanov, Davood Mohajerani and Marc Moreno Maza in [4].

1.1 Fürer's trick

Consider a prime field $\mathbb{Z}/p\mathbb{Z}$ and N, a power of 2, dividing $p-1$. Then, the finite field $\mathbb{Z}/p\mathbb{Z}$ admits an N-th primitive root of unity[1]. Denote such an element by ω. Let $f \in \mathbb{Z}/p\mathbb{Z}[x]$ be a polynomial of degree at most $N-1$. Then, computing the DFT of f at ω via an FFT produces the values of f at the successively powers of ω, that is, $f(\omega^0), f(\omega^1), \ldots f(\omega^{N-1})$. Using an asymptotically fast algorithm, namely a fast Fourier transform (FFT), this calculation amounts to:

1. $N \log(N)$ additions in $\mathbb{Z}/p\mathbb{Z}$,
2. $(N/2) \log(N)$ multiplications by a power of ω in $\mathbb{Z}/p\mathbb{Z}$.

If the size of p is k machine words, then

1. each addition in $\mathbb{Z}/p\mathbb{Z}$ costs $O(k)$ machine-word operations,
2. each multiplication by a power of ω costs $O(\mathsf{M}(k))$ machine-word operations,

where $n \longmapsto \mathsf{M}(n)$ is a multiplication time as defined in Section 2.3 Therefore, multiplication by a power of ω becomes a bottleneck as k grows. To overcome this difficulty, we consider the following trick proposed by Martin Fürer in [15, 16]. We assume that $N = K^e$ holds for some "small" K, say $K = 32$ and an integer $e \geq 2$. Further, we define $\eta = \omega^{N/K}$, with $J = K^{e-1}$ and assume that multiplying an arbitrary element of $\mathbb{Z}/p\mathbb{Z}$ by η^i, for any $i = 0, \ldots, K-1$, can be done within $O(k)$ machine-word operations. Consequently, every arithmetic operation (addition, multiplication) involved in a DFT on K points, using η as a primitive root, amounts to $O(k)$ machine-word operations. Therefore, such DFT of size K can be performed with $O(K \log(K)\, k)$ machine-word operations. instead of $O(K \log(K)\, \mathsf{M}(k))$ without the assumption. Since the multiplication time $n \longmapsto \mathsf{M}(n)$ is necessarily super-linear, the former estimate is asymptotically smaller than the latter one. As we shall see in Chapter 3, this result holds whenever p is a so called *generalized Fermat number*.

Returning to the DFT of size N at ω and using the factorization formula of Cooley and Tukey [6], we have

$$\mathrm{DFT}_{JK} \;=\; (\mathrm{DFT}_J \otimes I_K) D_{J,K} (I_J \otimes \mathrm{DFT}_K) L_J^{JK}, \tag{1.2}$$

see Section 5.1. Hence, the DFT of f at ω is essentially performed by:

1. K^{e-1} DFT's of size K (that is, DFT's on polynomials of degree at most $K-1$),
2. N multiplications by a power of ω (coming from the diagonal matrix $D_{J,K}$) and

[1] See Section 2.1.1 for this notion.

3. K DFT's of size K^{e-1}.

Unrolling Formula (1.2) so as to replace DFT_J by DFT_K and the other linear operators involved (the diagonal matrix D and the permutation matrix L) one can see that a DFT of size $N = K^e$ reduces to:

1. $e\,K^{e-1}$ DFT's of size K, and
2. $(e-1)\,N$ multiplications by a power of ω.

Recall that the assumption on the cost of a multiplication by η^i, for $0 \leq i < K$, makes the cost for one DFT of size K to $O(K\ \log_2(K)\,k)$ machine-word operations. Hence, all the DFT's of size K together amount to $O(e\,N\ \log_2(K)k)$ machine-word operations. That is, $O(N\ \log_2(N)\,k)$ machine-word operations. Meanwhile, the total cost of the multiplication by a power of ω is $O(e\,N\,\mathsf{M}(k))$ machine-word operations, that is, $O(N\ \log_K(N)\,\mathsf{M}(k))$ machine-word operations. Indeed, multiplying an arbitrary element of $\mathbb{Z}/p\mathbb{Z}$ by an arbitrary power of ω requires $O(\mathsf{M}(k))$ machine-word operations. Therefore, under our assumption, a DFT of size N at ω amounts to

$$O(N\ \log_2(N)\,k\ +\ N\ \log_K(N)\,\mathsf{M}(k)) \tag{1.3}$$

machine-word operations. When using generalized Fermat primes, we have $K = 2k$ and the above estimate becomes

$$O(N\ \log_2(N)\,k\ +\ N\ \log_k(N)\,\mathsf{M}(k)). \tag{1.4}$$

The second term in the big-O notation dominates the first one. Without our assumption, as discussed earlier, the same DFT would run in $O(N\ \log_2(N)\,\mathsf{M}(k))$ machine-word operations. Therefore, using generalized Fermat primes brings a speedup factor of $\log(K)$ w.r.t. the direct approach using arbitrary prime numbers.

In this thesis, we are addressing two questions. First, can we observe this speedup factor on a serial implementation written in the programming language C and run on modern multicore processors. Indeed, the authors of [4] answered a similar question in the case of a CUDA implementation targeting GPUs (Graphics Processing Units). Such architectures offer to programmers a finer control of hardware resources than multicore processors, thus more opportunities to reach high performance. Hence, this first question is a natural challenge.

Second, can we use FFT to implement multiplication in $\mathbb{Z}/p\mathbb{Z}$ and obtain better performance than using plain multiplication in $\mathbb{Z}/p\mathbb{Z}$? This was not attempted in the GPU implementation of [4]. However this is a natural question in the spirit of the algorithms of Schönhage and Strassen [25] and Fürer [15], where fast multiplication is achieved by "composing" FFTs operating on different vector sizes. The experimental results reported in Section 6 give positive answers to both questions.

1.2 Thesis organization and contributions

This thesis is organized as follows. Chapter 2 is a brief review of the concepts of a *prime field* and a *multiplication time*, the discrete Fourier transform, the fast Fourier Transform and its application to polynomial multiplication. Chapter 3 presents our implementation of prime fields of the form $\mathbb{Z}/p\mathbb{Z}$ where p is a Generalized Fermat prime number; this is based on and extends the work reported in [4].

Consider a Generalized Fermat prime number of the form $p = r^k + 1$, where k is a power of 2 and r is of machine-word size. As mentioned above, as well as in [4], multiplying by a power of

r modulo p can be done in $O(k)$ machine-word operations. However, multiplying two arbitrary elements of $\mathbb{Z}/p\mathbb{Z}$ is a non-trivial operation. Note that we encode elements of $\mathbb{Z}/p\mathbb{Z}$ in radix r expansion. Thus, multiplying two arbitrary elements of $\mathbb{Z}/p\mathbb{Z}$ requires to compute the product of two univariate polynomials in $\mathbb{Z}[X]$, of degree less than k, modulo $X^k + 1$. In [4], this is done by using plain multiplication, thus $\Theta(k^2)$ machine-word operations. In Chapter 4, we explain how to multiply two arbitrary elements x, y of $\mathbb{Z}/p\mathbb{Z}$ via FFT. We give a detailed analysis of the algebraic complexity of our procedure. A natural alternative to our approach would be to compute $(xy) \mod p$ where the product xy is an integer computed after converting the radix r expansion of x, y to integers (say in binary expansions). We show that this alternative approach is theoretically and practically less efficient than the one via FFT.

In order to verify experimentally the benefits of Fürer's trick, we need to perform FFT computations over a Generalized Fermat prime field $\mathbb{Z}/p\mathbb{Z}$, for different implementations of that prime field. One should be able to assume that the elements of $\mathbb{Z}/p\mathbb{Z}$ are in radix r expansion (when p writes $r^k + 1$ where k is a power of 2) or one should simply be able to use traditional radix 2 expansions. Moreover, we consider multiplying two arbitrary elements of $\mathbb{Z}/p\mathbb{Z}$ via FFT. Overall, we need an implementation of FFT running over a variety of prime fields. Chapter 5 reports on a generic implementation of FFT over finite fields in the BPAS library [3]. This part is a joint work with Colin Costello and Davood Mohajerani.

Finally, Chapter 6 gathers experimental results which yield positive answers to the research questions stated above. This part is also a joint work with Colin Costello and Davood Mohajerani.

Chapter 2

Background

2.1 Prime field arithmetic

Arithmetic operations for polynomials and matrices over prime fields play a central role in computer algebra. It supports the computation over Galois fields that are essential to cryptography algorithms as well as coding theory. In symbolic computation, the implementation of the so-called modular methods, prime fields are often using machine word size characteristic. Increasing the arithmetic to greater precision can be done using the Chinese Remainder Theorem (CRT).

However, using these small prime numbers can cause problems in some certain modular methods. In particular, the so-called *unlucky primes* are to be avoided [1, 8]. Because of the limitation of using small prime numbers, arithmetic over prime fields, where the primes are multi-precision numbers, is desired for some problems, for instance, polynomial system solving.

In algebra, a non-empty set $\mathbb{A}$ is a *ring* whenever $\mathbb{A}$ is endowed with two binary operations denoted additively and multiplicatively such that

- both addition and multiplication are associative,
- both addition and multiplication admit a neutral element, denoted respectively 0 and 1,
- addition must be commutative and every $x \in \mathbb{A}$ admits a symmetric element w.r.t. the addition, denoted $-x$.
- the multiplication is distributive w.r.t. the addition.

The ring $\mathbb{A}$ is *commutative* if its multiplication is commutative. The commutative ring $\mathbb{A}$ is a field if every non-zero $x \in \mathbb{A}$ admits a symmetric element w.r.t. the multiplication, denoted x^{-1}. Examples of fields are the set $\mathbb{Q}$ of rational numbers, the set $\mathbb{A}$ of real numbers and the set $\mathbb{C}$ of complex numbers. Examples of rings that are not fields are the set $\mathbb{Z}$ of integer numbers, the set of 2×2 matrices with coefficients in $\mathbb{R}$ and the set of univariate polynomials with coefficients in $\mathbb{Q}$.

A *Galois field*, also known as *finite field*, is a field with finitely many elements. The residue classes modulo p, where p is a prime number, form a field (unique up to isomorphism) called the *prime field* with p elements and denoted by $\mathsf{GF}(p)$ or $\mathbb{Z}/p\mathbb{Z}$. Single-precision and multi-precision primes are referred to as *small primes* and *big primes* respectively.

Let a, b be integers and p a prime number. The residue class of a in $\mathsf{GF}(p)$ is denoted by a

mod p. The sum and the product of $a \mod p$ and $b \mod p$ are given by $(a+b) \mod p$ and $a \cdot b \mod p$, respectively. If $b \mod p$ is not zero, then the quotient of $a \mod p$ by $b \mod p$ is given by $a \cdot b^{-1} \mod p$, where $b^{-1} \mod p$ is the inverse of b in $\mathsf{GF}(p)$. The element $b^{-1} \mod p$ can be computed in different ways, for instance via the Extended Euclidean Algorithm, see Chapter 5 in [17]. The maps $(a, b, p) \longmapsto a + b \mod p$, $(a, b, p) \longmapsto a \cdot b \mod p$ and $(a, b, p) \longmapsto ab^{-1} \mod p$ are often called *modular addition*, *modular multiplication* and *modular division*.

2.1.1 Primitive root of unity

Primitive roots of unity are a special kind of elements in a field that are used by some algorithms, such as Fast Fourier transforms. For a field $\mathbb{F}$ and an integer $n \geq 1$, an element $\omega \in \mathbb{F}$ is an *n-th primitive root of unity*, if it meets the following two requirements [17].

(*i*) ω is an *n-th root of unity*, that is, we have $\omega^n = 1$.
(*ii*) we have $\omega^i \neq 1$ for all $1 < i < n$.

This definition generalizes to the case where $\mathbb{F}$ is a commutative ring by adjusting the second requirement as follows: for all $1 < i < n$, the element $\omega^i - 1$ is not a zero-divisor.

It follows from a classical result in group theory, see [17], that the field $\mathsf{GF}(p)$ admits an n-th primitive root of unity if and only n divides $p - 1$. Assume from now on that that this latter condition holds. Then, we can derive a simple probabilistic algorithm to compute an n-th primitive root of unity in $\mathsf{GF}(p)$. By assumption, there exists an integer q such that $p = qn + 1$ holds. According to Fermat's little theorem, for all $a \in \mathsf{GF}(p)$ with $a \neq 1$ and $a \neq 0$, we have $a^{p-1} = 1$ which means $a^{qn} = 1$. This implies that a^q can be a candidate of n-th primitive root of unity. If a^q is not a n-th primitive root of unity, we would have $a^{qn/2} = 1$. Since $a^{qn/2} = -1$ or $a^{qn/2} = 1$ must hold, we know that if $a^{qn/2} = -1$ holds then a^q is a n-th primitive root of unity in $\mathsf{GF}(p)$. This trick is certainly well-known. As far as we know, it was first proposed by Xin Li in [12] and used in the `modpn` library [22].

Hence, we have Algorithm 2.1 for computing a n-th primitive root of unity in $\mathsf{GF}(p)$. In our implementation of finite field arithmetic in the BPAS library [3], this algorithm has always found an n-th primitive root of unity after a few tries and has never become a performance bottleneck.

Algorithm 2.1 Computing the n-th primitive root of unity over $\mathsf{GF}(p)$

input:
- a prime number p,
- an integer n which is a power of 2 dividing $p-1$.

output:
- an n-th primitive root of unity over $\mathsf{GF}(p)$

procedure PRIMITIVEROOTOFUNITY(p, n)
  $q := (p-1)/n$
  $d := qn/2$
  $c := 0$
  while $c^d \neq -1 \mod p$ **do**
    $c :=$ *randomnumber*()
  end while
  return c^d
end procedure

2.1.2 Montgomery multiplication

Montgomery multiplication is an algorithm for performing modular multiplication. It was presented by Peter L. Montgomery in 1985 [23]. This algorithm can speed up modular multiplication by avoiding division by the modulus without affecting modular addition and subtraction.

For a modulo p, let R be a number greater than p that is coprime to p. Assume also that R is some power of 2; hence multiplication and division by R can be done by shifting (on a computer using binary expansions for numbers); thus, they can be seen as inexpensive operations to perform. Since $\gcd(R, p) = 1$ holds, there exists a unique pair (R', p') of integers satisfying the following relation:

$$RR' - pp' = 1 \tag{2.1}$$

with $0 < R' < p$ and $0 < p' < R$. So that we have $p' = -p^{-1} \mod R$.

For a non-negative integer a, where $0 \leq a < Rp$, *Montgomery reduction* computes $c := aR^{-1} \mod p$ without division modulo p. Indeed, we have:

$$\begin{aligned} m &= ap' \mod R \quad \text{for } 0 \leq m < R \\ c &= (a + mp)/R \end{aligned} \tag{2.2}$$

if $c \geq p$ holds, then $c := c - p$ is performed.

To prove the correctness of $c = aR^{-1} \mod p$, firstly, we have $mp \equiv app' \equiv -a \mod R$, which means there exits an integer h such that $mp = hR - a$. Secondly, we have $c = (a+mp)/R = (a+hR-a)/R = h$, which means c is an integer. Also, $cR = a+mp \equiv a \mod p$, so that $c \equiv aR^{-1} \mod p$. Lastly, since $0 \leq a, mp < Rp$, we have $0 \leq a + mp < 2Rp$, which gives us $0 \leq c < 2p$, that is either $c = aR^{-1}$ or $c = aR^{-1} + p$ holds.

Let $x, y \in \mathsf{GF}(p)$. We "represent" x (or map x to) $\tilde{x} := xR \mod p$. Similarly, we represent y with $\tilde{y} := yR \mod p$. *Montgomery multiplication* uses Montgomery reduction on the representatives $\tilde{x}$ and $\tilde{y}$ of x and y. Indeed, we have:

$$\tilde{x}\,\tilde{y}R^{-1} = (xRyR)R^{-1} = (xy)R \mod p. \tag{2.3}$$

Hence, computing $(\tilde{x}\,\tilde{y})R^{-1}$ via Montgomery reduction produces the representative

$$\tilde{xy} = (xy)R \mod p \tag{2.4}$$

of xy. We observe that $x \tilde{+} y = \tilde{x} + \tilde{y} \mod p$ holds. Hence, $x \longmapsto \tilde{x}$ defines a 1-to-1 map from $\mathsf{GF}(p)$ to itself, which is:

1. compatible with addition, and
2. via Montgomery reduction, compatible with multiplication.

Therefore, if a sequence of arithmetic operations (addition, multiplication) is to be performed in $\mathsf{GF}(p)$, it can be advantageous to:

1. map the input $x, y, \ldots$ to $\tilde{x}, \tilde{y}, \ldots$,
2. compute with $\tilde{x}, \tilde{y}, \ldots$ instead of $x, y, \ldots$,
3. revert the mapping on the output.

This is the strategy that we follow with discrete Fourier transforms over prime fields.

2.2 The discrete Fourier transform and the fast Fourier Transform

Let $\mathbb{A}$ be a ring, and $\omega \in \mathbb{A}$ is an n-th primitive root of unity. The Discrete Fourier Transform (DFT) evaluates a univariate polynomial over $\mathbb{A}$ with degree at most n at the successive powers of ω. For a polynomial $f(x) = \sum_{i=0}^{n-1} f_i\, x^i \in \mathbb{A}[x]$, the Discrete Fourier Transform is defined as follows[17]:

Definition 1 *The $\mathbb{A}$-linear map*

$$DFT_\omega = \begin{cases} \mathbb{A}^n \to \mathbb{A}^n \\ f \to (f(1), f(\omega), \ldots, f(\omega^{n-2}), f(\omega^{n-1})) \end{cases}$$

*which evaluates a polynomial at the power of ω is called the **Discrete Fourier Transform** at ω.*

A ***Fast Fourier Transform*** (or FFT for short) is an algorithm which computes the DFT in an efficient way. FFTs were (re-)discovered by Cooley and Tukey [6] in 1965. We present below a popular example of the FFT, based on a 2-way divide-and-conquer strategy. Write $f(x) = \sum_{i=0}^{n-1} f_i\, x^i \in \mathbb{A}[x]$. Let q_0 and m_0 (resp. q_1 and m_1) be the quotient and the reminder of f divided by $x^{n/2} + 1$ (resp. $x^{n/2} - 1$). Hence, we have:

$$f = q_0\,(x^{n/2} - 1) + m_0 \tag{2.5}$$

and

$$f = q_1\,(x^{n/2} + 1) + m_1. \tag{2.6}$$

Note that since the degree of f is less than n, the degrees of q_0, q_1, m_0, m_1 are less than $n/2$. Observe that we compute q_0, q_1, m_0, m_1 easily. Indeed, let $A, B \in \mathbb{A}[x]$ be two polynomials with degrees less than $n/2$ and

$$f = A\, x^{n/2} + B \tag{2.7}$$

Then we can re-write Equation (2.5) and (2.6) as

$$f = A\,(x^{n/2} - 1) + B + A \tag{2.8}$$

and

$$f = A\,(x^{n/2} + 1) + B - A \tag{2.9}$$

Hence, we have

$$m_0 = B + A \tag{2.10}$$

and

$$m_1 = B - A \tag{2.11}$$

Now we use equation 2.5 to evaluate f at ω^{2i} for $0 \leq i < n/2$, we have

$$f(\omega^{2i}) = q_0(\omega^{2i})\,((\omega^{2i})^{n/2} - 1) + m_0(\omega^{2i}) = q_0(\omega^{2i})\,(\omega^{ni} - 1) + m_0(\omega^{2i}) = m_0(\omega^{2i}) \tag{2.12}$$

since $\omega^n = 1$.

Similarly we use equation 2.6 to evaluate f at ω^{2i+1} for $0 \leq i < n/2$

$$f(\omega^{2i+1}) = q_1(\omega^{2i+1})\,((\omega^{2i+1})^{n/2}+1)+m_1(\omega^{2i+1}) = q_1(\omega^{2i+1})\,(\omega^{ni}\,\omega^{n/2}+1)+m_1(\omega^{2i+1}) = m_1(\omega^{2i+1}) \tag{2.13}$$

since $\omega^{n/2} = -1$. Indeed, since ω is a n-th primitive root of unity, we have $\omega^n = (\omega^{n/2})^2 = 1$ and $\omega^{n/2} \neq 1$.

Now we can safely say that evaluating f at $(1, \omega, \ldots, \omega^{n-1})$ is the same as

- evaluating m_0 at ω^{2i} for $0 \leq i < n/2$
- and evaluating m_1 at ω^{2i+1} for $0 \leq i < n/2$

To make things simple, we define $m_1'(x) = m_1(\omega\, x)$ so that we can evaluate m_0 and m_1' at the same points that are all the even powers of ω.

The FFT algorithm is described as follow.

Algorithm 2.2 The Fast Fourier Transform

input:
- $n = 2^k \in \mathbb{N}$
- a polynomial $f = \sum_{i=0}^{n-1} f_i\, x^i \in \mathbb{A}$,
- $(1, \omega, \ldots, \omega^{n-1})$ powers of $\omega \in \mathbb{A}$ where ω is a n-th primitive root of unity.

output:
- $DFT_\omega = (f(1), f(\omega), \ldots, f(\omega^{n-1}))$

procedure FastFourierTransform(f, ω, n)
  if n = 1 **then**
    return f_0
  end if
  $m_0 := \sum_{i=0}^{n/2-1}(f_i + f_{i+n/2})x^i$
  $m_1' := \sum_{i=0}^{n/2-1}(f_i - f_{i+n/2})\omega^i\, x^i$
  call the algorithm recursively to evaluate m_0 and m_1' at the first $n/2$ powers of ω^2
  return $(m_0(1), m_1'(1), \ldots, m_0(\omega^{n-2}, m_1'(\omega^{n-2}))$
end procedure

2.3 Multiplication time

Throughout this thesis, we discuss many algorithms that are based on fast polynomial and integer multiplication algorithms. In order to simplify the notation for these multiplication algorithms in our analysis, we follow the definition of ***multiplication time*** in [17](Definition 8.26) that is:

Definition 2 *Let* $\mathsf{M} : \mathbb{N} \to \mathbb{R}$ *be a function satisfying* $\mathsf{M}(n) \geq n$ *and* $\mathsf{M}(m+n) \geq \mathsf{M}(m) + \mathsf{M}(n)$ *for all* $n, m \in \mathbb{N}$. *We say that* $\mathsf{M} : \mathbb{N} \to \mathbb{R}$ *is a multiplication time for polynomials if, for every commutative ring* $\mathbb{A}$*, for every non-negative integer n, any two polynomials in* $\mathbb{A}[x]$ *of degree less than n can be multiplied using at most* $\mathsf{M}(n)$ *operations in* $\mathbb{A}$*. Similarly, we say that* $\mathsf{M} : \mathbb{N} \to \mathbb{R}$ *is a multiplication time for integers if, for non-negative integer n, any two integers of bit-size less than n can be multiplied using at most* $\mathsf{M}(n)$ *word operations.*

In the next section, we give multiplication times based on well-known polynomial multiplication algorithms.

2.4 Dense univariate polynomial multiplication

Multiplication between dense univariate polynomials is a widely used procedure in computer algebra. As the degree of the polynomials increase, the complexity grows significantly such that different fast multiplication algorithms are proposed for polynomials with different features. Here we give a brief introduction on these algorithms and the comparison among them.

Classical algorithm

We have learned the most classic and naive polynomial multiplication algorithm in public school. It is given by the definition of polynomial multiplication. For two polynomials $f(x)$ and $g(x)$, we simply multiply each term in $f(x)$ with each term in $g(x)$ and use addition to normalize the final result. Given $deg(f) = n$ and $deg(g) = m$, we have the following general equation

$$f(x) \cdot g(x) = \sum_{i=0}^{n-1} \sum_{j=0}^{m-1} f_i \, g_j \, x^{i+j} \tag{2.14}$$

we need mn multiplications and mn additions in total. Hence, to multiply two polynomials with degree less than n, we have the multiplication time of the classic algorithm as

$$\mathsf{M}(n) = O(n^2) \tag{2.15}$$

Karatsuba's algorithm

Karatsuba's algorithm is a fast multiplication algorithm, discovered in 1962 by Karatsuba [21]. It increases the total number of operations but using more addition and subtraction to reduce

the number of multiplications. For example, we want to compute $f = a + b$ times $g = c + d$, using the classic multiplication we have

$$f\,g = ac + bc + ad + bd \tag{2.16}$$

which requires four multiplication and three addition operations. But if using the following method

$$f\,g = ac + e + bd,\ e = (a + b)(c + d) - ac - bd = bc + ad \tag{2.17}$$

we only need three multiplications and some additions and subtractions. Since multiplication is more expensive than addition and subtraction, the total cost decreases.

Now, let's say we have two polynomials $f = \sum_{i=0}^{n-1} f_i\,x^i$ and $g = \sum_{j=0}^{n-1} g_j\,x^j$ with degree less than n. We rewrite them using the following representation

$$f = F_1\,x^{n/2} + F_0 \tag{2.18}$$

$$g = G_1\,x^{n/2} + G_0 \tag{2.19}$$

with the degree of F_1, F_0, G_1, G_0 less than $n/2$. We compute f times g by

$$f\,g = F_1\,G_1\,x^n + ((F_1 + F_0)\,(G_1 + G_0) - F_1\,G_1 - F_0\,G_0)\,x^{n/2} + F_0\,G_0 \tag{2.20}$$

We only need three multiplications of polynomials with degree less than $n/2$ and some additions. By using the above equation recursively on smaller degrees, we will save the total cost significantly.

Here is an algorithm using this idea.

Algorithm 2.3 Karatsuba Multiplication

input:
- $n = 2^k \in \mathbb{N}$
- two polynomials $f, g \in \mathbb{A}[x]$ with degree less than n where $\mathbb{A}$ is a commutative ring with 1.

output:
- $f * g \in \mathbb{A}[x]$

procedure KaratsubaMultiplication(f, g, n)
  Let $f = F_1\,x^{n/2} + F_0$ with $deg(F_1), deg(F_0) < n/2$
  Let $g = G_1\,x^{n/2} + G_0$ with $deg(G_1), deg(G_0) < n/2$
  $F := F_1 + F_0$
  $G := G_1 + G_0$
  Compute $F_1\,G_1$, $F\,G$ and $F_0\,G_0$ by calling this procedure recursively
  return $F_1\,G_1\,x^n + (F\,G - F_1\,G_1 - F_0\,G_0)\,x^{n/2} + F_0\,G_0$
end procedure

Algorithm 2.3 computes the multiplication between two polynomials with degree less than n with at most $9n^{\log 3}$ operations over a ring (see [17] Section 8.1 for more details). Hence, the multiplication time of Karatsuba's algorithm is

$$\mathsf{M}(n) = O(n^{\log 3/\log 2}) = O(n^{1.59}) \tag{2.21}$$

There is a further generalization of Karatsuba's algorithm, known as Toom-Cook algorithm, that can be even faster with n large enough, which splits the polynomials into k parts, for $k \geq 3$ and $k \in \mathbb{Z}$.

FFT-based algorithm

To multiply two polynomials with degree less than n on a ring $\mathbb{A}[x]$, the convolution *w.r.t* n is commonly used[17].

Definition 3 *The convolution w.r.t n of polynomials* $f = \sum_{i=0}^{n-1} f_i\, x^i$ *and* $g = \sum_{j=0}^{n-1} g_j\, x^j$ *is*

$$h = \sum_{k=0}^{n-1} h_k\, x^k \tag{2.22}$$

where each h_k *is*

$$h_k = \sum_{i+j \equiv k \mod n} f_i\, g_j \tag{2.23}$$

We use $f *_n g$ *to represent the convolution of two polynomials with degree less than n. If n is clear from content we can simple use* $f * g$ *instead.*

Notice that $f * g \equiv f(x)\, g(x) \mod (x^n - 1)$, which means that the convolution of f and g in ring $\mathbb{A}[x]$ is equivalent to multiplying f and g in $\mathbb{A}[x]/\langle x^n - 1\rangle$

We know from [17] that

$$DFT_\omega(f * g) = DFT_\omega(f)\, DFT_\omega(g) \tag{2.24}$$

hence, the convolution of two polynomials can be computed using the following algorithm.

Algorithm 2.4 Fast Convolution

1: **input:**
   - $n = 2^k \in \mathbb{N}$
   - two polynomials $f, g \in \mathbb{A}[x]$ with degree less than n,
   - a n-th primitive root of unity $\omega \in \mathbb{A}$.
2: **output:**
   - $f * g \in \mathbb{A}[x]$
3: **procedure** FASTCONVOLUTION(f, g, ω, n)
4: compute the first n powers of ω
5: $\alpha := DFT_\omega(f)$
6: $\beta := DFT_\omega(g)$
7: $\gamma := \alpha\beta$ ▷ Component-wise multiplication
8: **return** $(DFT_\omega)^{-1}(\gamma) := \frac{1}{n} DFT_{\omega^{-1}}(\gamma)$
9: **end procedure**

Recall from Section 2.2 that the Fast Fourier Transform can compute the Discrete Fourier Transform quickly. Figure 2.1 shows the FFT-based polynomial multiplication in $\mathbb{A}[x]/\langle x^n - 1\rangle$.

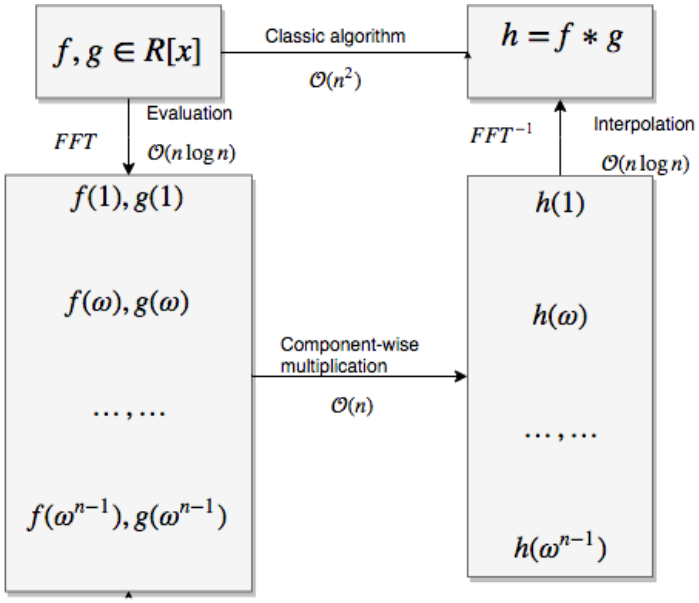

Figure 2.1: FFT-based univariate polynomial multiplication

The whole procedure needs $\frac{9}{2}n \log n + O(n)$ arithmetic operations in $\mathbb{A}$ (see [17] Theorem 8.18 for details) so that the multiplication time of FFT-based multiplication is

$$\mathsf{M}(n) = O(n \log n) \tag{2.25}$$

Table2.1 gives the multiplication time for some popular fast multiplication algorithms.

Table 2.1: Multiplication time of different algorithms.

Algorithm	$\mathsf{M}(n)$
Classic Algorithm	$O(n^2)$
Karatsuba's Algorithm	$O(n^{1.585})$
Toom-3	$O(n^{1.465})$
Toom-4	$O(n^{1.404})$
FFT-based Algorithm	$O(n \log n)$
Schönhage-Strassen's Algorithm	$O(n \log n \log \log n)$

2.5 Big O and Θ notations

When analyzing the complexity of algorithms, we use the big O and the Θ notations, where big O gives a asymptotic upper bound of a function and Θ gives an order of magnitude of a function.

Let f and g be functions from $\mathbb{N}$ to $\mathbb{R}$.

Definition 4 *We say that $g(n)$ is in the **order of magnitude** of $f(n)$ and write $f(n) \in \Theta(g(n))$ if there exists two strictly positive constants c_1 and c_2 such that for n big enough we have*

$$0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$$

Definition 5 *We say that $g(n)$ is an **asymptotic upper bound** of $f(n)$ and write $f(n) \in O(g(n))$ if there exists a strictly positive constant c_2 such that for n big enough we have*

$$0 \leq f(n) \leq c_2 g(n)$$

The big O (and Θ) can also be used with multiple variables as follows. Let f, g be functions, with positive real values, and depending on a vector $\vec{n} = (n_1, \ldots, n_q)$ of non-negative integers. We write $f(\vec{n}) \in O(g(\vec{n}))$ whenever there exits two strictly positive constant M and C, such that for all $\vec{n}$ satisfying $n_i \geq M$ for all $1 \leq i \leq q$ we have $0 \leq f(\vec{n}) \leq Cg(\vec{n})$ [7].

For us the multivariate version of big O given above is too strong. We define the multivariate big O as follows for two integer variables w, n.

Definition 6 *$f(w, n) \in O(g(w, n))$ means that there exist three positive integer constants w_1, w_2, n_1, with $w_1 < w_2$, and a positive real constant c such that for all positive integers w, n, if $w_1 \leq w \leq w_2$ and $n \geq n_1$ both hold, then we have $f(w, n) \leq cg(w, n)$.*

2.6 Syntax of pseudo-code

We use the following syntax in all the pseudo-code of the algorithms:

- $a := b$ assigns value b to variable a
- $a = b$ returns true if a is equal to b, otherwise returns false
- $\vec{x}$ is a vector
- x_i is the i-th element in $\vec{x}$

Here are some notations for the C code we present in the paper:

- `usfixn64` is the type of unsigned 64-bit integer
- `sfixn` is the type of signed 64-bit integer
- `U64_MASK` is defined as $2^{64} - 1$
- `U128_MASK` is defined as $2^{128} - 1$

Chapter 3

Generalized Fermat prime field arithmetic

Small prime field arithmetic has been implemented in different computer algebra systems. With the help of tricks like Montgomery's reduction, this can be done efficiently, but the small characteristic restricts the precision to a single machine word. Multi-precision numbers can be handled using the Chinese Remainder Theorem. Nevertheless, for certain algorithms in computer algebra, like modular methods for polynomial systems [1, 8, 2] it is desirable to use prime fields of large characteristic, thus computing modulo prime numbers with size on the order of several machine words.

Since modular methods for polynomial systems rely on polynomial arithmetic, those large prime numbers must support FFT-based algorithms, such as FFT-based polynomial multiplication. This leads us to consider the so-called Generalized Fermat prime numbers.

The n-th Fermat number can be denoted by $F_n = 2^{2^n} + 1$. This sequence of numbers plays an essential role in number theory. Arithmetic operations on fields based on Fermat numbers are simpler than those of other arbitrary prime numbers since 2 is the 2^{n+1}-th primitive root of unity modulo F_n. But, unfortunately, the largest Fermat prime number known now is F_4. This triggered the interests of finding Fermat-like numbers. *Generalized Fermat numbers* are one of these kinds.

Numbers that are in the form of $a^{2^n} + b^{2^n}$ with a, b any co-prime integers, where $a > b > 0$ and $n > 0$ hold, are called *generalized Fermat numbers*. Among all, those with $b = 1$ are of the most interest; we commonly write generalized Fermat numbers of the form $a^{2^n} + 1$ as $F_n(a)$. For a generalized Fermat number p, we use $\mathbb{Z}/p\mathbb{Z}$ to represent the finite field $\mathsf{GF}(p)$. In particular, in the field $\mathbb{Z}/F_n(a)\mathbb{Z}$, a is a 2^{n+1}-th primitive root of unity. But with the binary representation of numbers on computers, the arithmetic operations on such fields are not as simple as those of Fermat numbers. To solve this problem, a special kind of generalized Fermat number is defined in the previous work of our research group [4].

Any integer in the form of $F_n(r) = (2^w \pm 2^u)^k + 1$ is called a sparse radix generalized Fermat number, where $w > u \geq 0$. Table 3.1 lists some sparse radix generalized Fermat numbers that are primes. For each prime $p = F_n(r)$, k is some power of 2 and the prime writes as $p = r^k + 1$. In the same table, the number 2^e is the largest power of 2 that divides $p - 1$, which gives the maximum length of a vector to which we can apply a 2-way FFT algorithm.

In Section 3.1, we will introduce how we can use the radix-r representation to represent the elements in $\mathbb{Z}/p\mathbb{Z}$. Section 3.2 will introduce the special primitive roots of unity that can benefit us when computing FFT over $\mathbb{Z}/p\mathbb{Z}$, and will give an algorithm on how to get those primitive

Table 3.1: SRGFNs of practical interest.

p	$\max\{2^e \text{ s.t. } 2^e \mid p-1\}$
$(2^{63}+2^{53})^2+1$	2^{106}
$(2^{64}-2^{50})^4+1$	2^{200}
$(2^{63}+2^{34})^8+1$	2^{272}
$(2^{62}+2^{36})^{16}+1$	2^{576}
$(2^{62}+2^{56})^{32}+1$	2^{1792}
$(2^{63}-2^{40})^{64}+1$	2^{2560}
$(2^{64}-2^{28})^{128}+1$	2^{3584}

roots. Next, in Section 3.3, we will show the algorithm for doing addition and subtraction in $\mathbb{Z}/p\mathbb{Z}$. In Section 3.4, we will discuss how we can multiply any elements with a power of r efficiently. This idea fits in Fürer's algorithm that, for DFT on certain points, multiplication by the particular primitive roots can be done in a very cheap way. Last, in Section 3.5, we give the basic algorithm on multiplication between two arbitrary elements in $\mathbb{Z}/p\mathbb{Z}$.

3.1 Representation of $\mathbb{Z}/p\mathbb{Z}$

In the finite prime field $\mathbb{Z}/p\mathbb{Z}$, where $p = r^k + 1$, each element x is represented by a vector $\vec{x} = (x_{k-1}, \ldots, x_0)$ of length k. We restrict all the coefficients to be non-negative integers so that we have

$$X \equiv x_{k-1}\, r^{k-1} + x_{k-2}\, r^{k-2} + \cdots + x_1\, r + x_0 \mod p \tag{3.1}$$

The following two cases make the representation unique for each element:

1. When $x \equiv p-1 \mod p$ holds, we have $x_{k-1} = r$ and $x_{k-2} = \cdots = x_0 = 0$.
2. When $0 \le x < p-1$ holds, we have $0 \le x_i < r$ for $i = 0, \ldots, k-1$.

We can also use a univariate polynomial $f_x \in \mathbb{Z}[R]$ to represent x: we write $f_x = \sum_{i=0}^{k-1} x_i R^i$, such that $x \equiv f_x(r) \mod p$.

When computing the representation $\vec{x}$ of a number $x < p$, the case where $x = p-1$ is trivial, since we can directly set x_{k-1} to r and $x_{k-2}, \ldots, x_0$ to 0. Consider the case $0 \le x < p-1$. Let $q_k = 0$ and $s_k = x$. Then, for $0 \le i < k$, q_i and s_i are the quotient and the remainder in the Euclidean division of s_{i+1} by r^i so that we have $x_i = q_i$, for $0 \le i < k$.

3.2 Computing the primitive root of unity in $\mathbb{Z}/p\mathbb{Z}$

Recall that for any n that divides $p-1$, we can use algorithm 2.1 to find an n-th primitive root of unity in $\mathbb{Z}/p\mathbb{Z}$. Now we want to consider the case of finding an N-th primitive root of unity ω in $\mathbb{Z}/p\mathbb{Z}$ such that $\omega^{N/2k} = r$ holds. Indeed, computing a DFT at such ω on a vector of size N would take advantage of the fact that multiplying by a power of r can be done in linear time, see Section 3.4.

In Algorithm 3.1, the input N is a power of 2 that divides $p-1$ and the input g is a N-th primitive root of unity in $\mathbb{Z}/p\mathbb{Z}$.

Algorithm 3.1 Primitive N-th root $\omega \in \mathbb{Z}/p\mathbb{Z}$ such that $\omega^{N/2k} = r$

```
1:  procedure BigPrimeFieldPrimitiveRootOfUnity(N, r, k, g)
2:      a := g^{N/2k}
3:      b := a
4:      j := 1
5:      while b ≠ r do
6:          b := a b
7:          j := j + 1
8:      end while
9:      ω := g^j
10:     return (ω)
11: end procedure
```

From the definition of generalized Fermat prime numbers we know that r is a $2k$-th primitive root of unity in $\mathbb{Z}/p\mathbb{Z}$, where $p = r^k + 1$. While $g^{N/2k}$ is a $2k$-th root of unity, it must equal to some power of r, say $r^t \mod p$ for some $0 \leq t < 2k$. Let j be a non-negative integer, q and s are the quotient and the remainder of j in the Euclidean division by $2k$, so we have

$$j = q \cdot 2k + s \tag{3.2}$$

and

$$g^{jN/2k} = g^{2kq+s}\, g^{N/2k} = g^s\, g^{N/2k} = (g^{N/2k})^s = r^{ts} \tag{3.3}$$

By the definition of primitive root of unity, the powers r^{ts} are pairwise different for $0 \leq s < 2k$ and for some s_i, $r^{ts_i} = r$ holds. Hence, for some $j_i = q_i \cdot 2k + s_i$, we will have $(g^{N/2k})^{j_i} = r$. Then $\omega = g^{j_i}$ is the primitive root of unity that we want.

3.3 Addition and subtraction in $\mathbb{Z}/p\mathbb{Z}$

Let $x, y \in \mathbb{Z}/p\mathbb{Z}$ represented by vectors $\vec{x}$ and $\vec{y}$. The following algorithm 3.2 computes $\overrightarrow{x+y}$ that represents the sum of x and y in $\mathbb{Z}/p\mathbb{Z}$. We firstly compute the component-wise addition of $\vec{x}$ and $\vec{y}$ with carry. If there's no carry beyond the last component u_{k-1}, then $u_0, \ldots, u_{k-1}$ is the vector representation of $x + y$ in $\mathbb{Z}/p\mathbb{Z}$. If there is a carry, then the sum is over r^k and we need to do a subtraction of carry by the vector $\vec{u}$, since $r^k \equiv p - 1 \equiv -1 \mod p$.

Algorithm 3.2 Computing $x + y \in \mathbb{Z}/p\mathbb{Z}$ for $x, y \in \mathbb{Z}/p\mathbb{Z}$

1: **procedure** BIGPRIMEFIELDADDITION($\vec{x}, \vec{y}, r, k$)
2: compute $z_i = x_i + y_i$ in $\mathbb{Z}/p\mathbb{Z}$, for $i = 0, \ldots, k-1$,
3: let $z_k = 0$,
4: for $i = 0, \ldots, k-1$, compute the quotient q_i and the remainder s_i in the Euclidean division of z_i by r, then replace (z_{i+1}, z_i) by $(z_{i+1} + q_i, s_i)$,
5: if $z_k = 0$ then return $(z_{k-1}, \ldots, z_0)$,
6: if $z_k = 1$ and $z_{k-1} = \cdots = z_0 = 0$, then let $z_{k-1} = r$ and return $(z_{k-1}, \ldots, z_0)$,
7: let i_0 be the smallest index, $0 \leq i_0 \leq k$, such that $z_{i_0} \neq 0$, then let $z_{i_0} = z_{i_0} - 1$, let $z_0 = \cdots = z_{i_0-1} = r - 1$ and return $(z_{k-1}, \ldots, z_0)$.
8: **end procedure**

In this theoretical algorithm, we use a Euclidean division to compute the carry and the remainder of $x_i + y_i$, which requires a division and a subtraction operation. But in practical implementation, we can avoid the expensive division. The following lists the C code we used in the BPAS library.

According to the method in 3.1, each component of $\vec{x}$ and $\vec{y}$ is in the range of $[0, k-1]$ for $0 \leq i \leq k-2$ and $x_{k-1}, y_{k-1} \in [0, k]$, so that we can safely say that the results of the component-wise addition will not be greater than $2r - 2$ for the first $k - 1$ pairs of component. Hence, if the sum is greater than r, we can simply subtract the result by r and set the carry to 1, instead of using an Euclidean division. For the last pair x_{k-1} and y_{k-1}, the two special cases are one of them is equal to r and both of them are equal to r. For the first case, the maximum sum of x_{k-1} and y_{k-1} is $2r - 1$, there is no difference from the previous method.

Now, let's consider the second case where both x_{k-1} and y_{k-1} are equal to r. And all of the other components in the vectors are 0, such that both x and y are equal to r^k. When we add the two components together u_{k-1} is equal to $2r$ and by using line 9 to 11 from listing 3.1, we have $u_{k-1} = r$ and carry = 1. Then u_{k-1} is the first u_i that is not 0. In line 33, we have $u_{k-1} = u_{k-1} - 1 = r - 1$ and in line 31 we set $u_i = r - 1$ for $0 \leq i < k - 1$. The result we get is $u_i = r - 1$ for $0 \leq i < k$, that is equal to $u \equiv -2 \mod p$, indeed that $x + y \equiv r^k + r^k \equiv 2(p-1) \equiv 2p - 2 \equiv -2 \mod p$. So far we have proved that our algorithm works correctly and efficiently for all of the cases.

```
1  sfixn* addition (sfixn * x, sfixn *y, int k, sfixn r) {
2          short c = 0;
3          short post = 0;
4          sfixn sum = 0;
5          int i = 0;
6
7          for (i=0;i < k;i++) {
8                  sum = x[i] + y[i] + c;
9                  if (sum >= r ) {
10                         c = 1;
11                         x[i] = sum - r;
12                 }
13                 else {
14                         x[i] = sum;
15                         c = 0;
16                 }
```

```
17          }
18
19          if (c > 0){
20                  post = -1;
21
22                  for (i = 0; i < k; i++) {
23                          if (x[i] != 0){
24                                  post = i;
25                                  break;
26                          }
27                  }
28
29                  if (post >= 0){
30                          for (i = 0; i < post; i++) {
31                                  x[i] = r - 1;
32                          }
33                          x[post]--;
34                  }
35                  else {
36                          x[k-1] = r;
37                          for (i = 0;i < k-1; i++){
38                                  x[i] = 0;
39                          }
40                  }
41          }
42
43          return x;
44  }
```

Listing 3.1: Addition in a Generalized Fermat Prime Field

Similarly, we have an algorithm BigPrimeFieldSubtraction$(\vec{x}, \vec{y}, r, k)$ for computing $\overleftrightarrow{x - y}$ represents $(x - y) \in \mathbb{Z}/p\mathbb{Z}$.

3.4 Multiplication by power of r in $\mathbb{Z}/p\mathbb{Z}$

Multiplication between two arbitrary elements in $\mathbb{Z}/p\mathbb{Z}$ can be very complicated and expensive, and Chapter 4 will explain that process in greater detail. Now, let us consider the case of multiplication between elements $x, y \in \mathbb{Z}/p\mathbb{Z}$, where one of them is a power of r. We assume that $y = r^i$ for some $0 \leq i \leq 2k$. The cases that $i = 0$ and $i = 2k$ are trivial, since r is a $2k$-th primitive root of unity in $\mathbb{Z}/p\mathbb{Z}$, we have $r^0 = r^{2k} = 1$. Also we have $r^k = -1$ in $\mathbb{Z}/p\mathbb{Z}$, so that for $i = k$, we have $x = -x$ and for $k < i < 2k$, $r^i = -r^{i-k}$ holds. Now let us only consider the

case that $0 < i < k$, we have the following equation:

$$\begin{aligned} xr^i &\equiv (x_{k-1}\, r^{k-1+i} + \cdots + x_0\, r^i) \mod p \\ &\equiv \sum_{j=0}^{j=k-1} x_j r^{j+i} \mod p \\ &\equiv \sum_{h=i}^{h=k-1+i} x_{h-i} r^h \mod p \\ &\equiv (\sum_{h=i}^{h=k-1} x_{h-i} r^h - \sum_{h=k}^{h=k-1+i} x_{h-i} r^{h-k}) \mod p \end{aligned}$$

We see that for all $0 \leq i \leq 2k$, $x \cdot r^i$ is reduced to some shift and a subtraction. We call this process cyclic shift. The following gives the C implementation in the BPAS library.

```
sfixn* MulPowR(sfixn *x,int s, int k, sfixn r){
        sfixn *a =(sfixn*)calloc(sizeof(sfixn),k);
        sfixn *b =(sfixn*)calloc(sizeof(sfixn),k);
        sfixn *c =(sfixn*)calloc(sizeof(sfixn),k);
        s = s%(2 * k);
        if (s == 0)
                return x;
        else if (s == k)
                return BigPrimeFieldSubtraction(c,x,k,r);
        else if ((s > k) && (s < (2 * k))){
                s = s - k;
                x = BigPrimeFieldSubtraction(c,x,k,r);
        }
        int i;
        for (i = 0; i < (k - s); i++)
                b[i + s] = x[i];
        for (i = k - s; i < k; i++)
                a[i - (k - s)] = x[i];
        if(x[k-1] == r){
                a[s-1] -=r;
                a[s] ++;
        }
        return BigPrimeFieldSubtraction(b,a,k,r);
}
```

Listing 3.2: Multiplication by power if r in a Generalized Fermat Prime Field

3.5 Multiplication between arbitrary elements in $\mathbb{Z}/p\mathbb{Z}$

According to Section 3.1, we use the univariate polynomials $f_x, f_y \in \mathbb{Z}[R]$ to represent input elements $x, y \in \mathbb{Z}/p\mathbb{Z}$ respectively. Algorithm 3.3 computes the product $x \cdot y \in \mathbb{Z}/p\mathbb{Z}$. In the first step, we multiply the two polynomials over $\mathbb{Z}$ and compute the remainder f_u of the product modulo $R^k + 1$. Then, we convert all the coefficients of f_u into the radix-r representation in $\mathbb{Z}/p\mathbb{Z}$. Finally we multiply each coefficient with the corresponding power of r using the "cyclic shift" operation from Section 3.4 and add all the results together.

Algorithm 3.3 Computing $x \cdot y \in \mathbb{Z}/p\mathbb{Z}$ for $x, y \in \mathbb{Z}/p\mathbb{Z}$

1: **input:**
   - an integer `k` and radix `r`,
   - two polynomials f_x and f_y whose coefficient vectors are $\vec{x}, \vec{y}$.
2: **output:**
   - a vector $\vec{u}$
3: **procedure** BigPrimeFieldMultiplication(f_x, f_y, r, k)
4: $f_u(R) := f_x(R) \cdot f_y(R)$ ▷ computing f_x times f_y in $\mathbb{Z}[R]$
5: $f_u(R) := f_u(R) \mod (R^k + 1)$ ▷ we get $f_u(R) = \sum_{i=0}^{k-1} u_i \cdot R^i$
6: $\vec{u}$ is the coefficient vector of f_u
7: **for** $0 \leq i < k$ **do**
8: $\vec{u_i} := u_i \in \mathbb{Z}/p\mathbb{Z}$ ▷ compute a radix representation of each u_i using method in 3.1
9: **end for**
10: $\vec{u} := \vec{u_0}$ ▷ add all the u_i together using the algorithm 3.2
11: **for** $1 \leq i < k$ **do**
12: $\vec{u}$:=BigPrimeFieldAddition($\vec{u}, \vec{u_i}, k, r$)
13: **end for**
14: return $\vec{u}$
15: **end procedure**

In the following chapter, we will discuss the multiplication between arbitrary elements in more detail, and we analyze the different implementations of the algorithm.

Chapter 4

Optimizing multiplication in Generalized Fermat prime fields

In this chapter, we will discuss how to multiply two arbitrary elements in $\mathbb{Z}/p\mathbb{Z}$ efficiently using FFT, when p is a Generalized Fermat prime. Firstly, in Section 4.1, we outline two algorithms that we can use for this multiplication: one is based on polynomial multiplication (see Section 4.1.1) and the other one is based on integer multiplication by means of the GMP library [18] (see Section 4.1.2). Then, in Section 4.2 we provide detailed complexity analysis on the two approaches. Finally, in Section 4.3, we present the implementation of the FFT-based polynomial-based multiplication. We break down the algorithm into sub-routines and explain in details for each part. The C functions that we use can be found in Appendix A.

4.1 Algorithms

Let p be a Generalized Fermat prime. When actually implementing the multiplication of two arbitrary elements in the field $\mathbb{Z}/p\mathbb{Z}$, we use two different approaches. In the first approach, we follow the basic idea explained in Chapter 3 (see Algorithm 3.3) which treats any two elements x, y in the field as polynomials f_x, f_y and uses polynomial multiplication algorithms to compute the product xy. The other approach involves converting the elements x, y from their radix-r representation into GMP integer numbers and letting the GMP library [18] do the job.

4.1.1 Based on polynomial multiplication

In Section 3.5 we gave the basic algorithm for multiplying two arbitrary elements of $\mathbb{Z}/p\mathbb{Z}$ based on polynomial multiplication. In practice, there are more details to be considered in order to reach high-performance. For instance, how do we efficiently convert a positive integer in the range (o, r^3) into radix-r representation.

Let us consider how to calculate $u = x\,y \mod p$ with $x, y, u \in \mathbb{Z}/p\mathbb{Z}$. Here we want to use the polynomial representation of the elements in the field, that is, $f_x(R) = x_{k-1}\,R^{k-1} + \cdots + x_1\,R + x_0$ and $f_y(R) = y_{k-1}\,R^{k-1} + \cdots + y_1\,R + y_0$. The first step is to multiply the two polynomials f_x and f_y. We can use different polynomial multiplication algorithms depending on the value of

k. Let us look at the expansion of f_u. Recall that taking a polynomial modulo by $R^n + 1$ means replacing every occurrence of R^n by -1.

$$\begin{aligned}
f_u(R) &= f_x(R) \cdot f_y(R) \mod (R^k + 1) \\
&= \sum_{m=0}^{2k-2} \sum_{0 \le i,j < k}^{i+j=m} x_i\, y_j\, R^m \mod (R^k + 1) \\
&= (x_{k-1}\, y_0 + x_{k-2}\, y_1 + x_{k-3}\, y_2 + \cdots + x_1\, y_{k-2} + x_0\, y_{k-1})\, R^{k-1} \\
&+ (x_{k-2}\, y_0 + x_{k-3}\, y_1 + \cdots + x_1\, y_{k-3} + x_0\, y_{k-2} - x_{k-1}\, y_{k-2})\, R^{k-2} \\
&+ (x_{k-3}\, y_0 + x_{k-4}\, y_1 + \cdots + x_0\, y_{k-3} - x_{k-1}\, y_{k-2} - x_{k-2}\, y_{k-1})\, R^{k-2} \\
&\quad \cdots \\
&+ (x_1\, y_0 + x_0\, y_1 - x_{k-1}\, y_2 - \cdots - x_2\, y_{k-1})\, R \\
&+ (x_0\, y_0 - x_{k-1}\, y_1 - \cdots - x_1\, y_{k-1}) \\
&= \sum_{m=0}^{k-1} (\sum_{0 \le i,j < k}^{i+j=m} x_i\, y_j - \sum_{0 \le i,j < k}^{i+j=k+m} x_i\, y_j)\, R^m
\end{aligned}$$

Each coefficient u_i of f_u is the combination of k monomials, so the absolute value of each u_i is bounded over by $k \cdot r^2$ which implies that it needs at most $\lfloor \log k + 2 \log r \rfloor + 1$ bits to be encoded. Since k is usually between 4 to 256, a radix r representation of u_i of length 3 is sufficient to encode u_i. Hence, we denote by $[c_i, h_i, l_i]$ the 3 integers uniquely given by:

1. $u_i = c_i r^2 + h_i r + l_i$,
2. $0 \le h_i, l_i < r$.
3. $c_i \in [-(k-1), k]$,
4. $c_i u_i \ge 0$ holds.

Then, we can rewrite:

$$\begin{aligned}
f_u(R) &= f_x(R) \cdot f_y(R) \mod (R^k + 1) \\
&= (c_0 R^2 + h_0 R + l_0) + (c_1 R^2 + h_1 R + l_1) R + (c_2 R^2 + h_2 R + l_2) R^2 + \cdots \\
&\quad + (c_{k-2} R^2 + h_{k-2} R + l_{k-2}) R^{k-2} + (c_{k-1} R^2 + h_{k-1} R + l_{k-1}) R^{k-1} \\
&= \sum_{i=0}^{k-1} (c_i\, R^{2+i} + h_i\, R^{1+i} + l_i\, R^i)
\end{aligned}$$

Now we obtain three vectors $\vec{c} = [c_0, c_1, \ldots, c_{k-1}]$, $\vec{h} = [h_0, h_1, \ldots, h_{k-1}]$ and $\vec{l} = [l_0, l_1, \ldots, l_{k-1}]$ with k coefficients each. As we shift $\vec{c}$ to the right twice and $\vec{h}$ to the right once, we deduce three numbers c, h, l in the radix-r representation.

$$\begin{aligned}
c &= c_{k-3}\, r^{k-1} + c_{k-4}\, r^{k-2} + \cdots + c_0\, r^2 + c_{k-2}\, r + c_{k-1} \\
h &= h_{k-2}\, r^{k-1} + h_{k-3}\, r^{k-2} + \cdots + h_1\, r^2 + h_0\, r + h_{k-1} \\
l &= l_{k-1}\, r^{k-1} + l_{k-2}\, r^{k-2} + \cdots + l_2, r^2 + l_1\, r + l_0
\end{aligned}$$

At last we need two additions in $\mathbb{Z}/p\mathbb{Z}$ to compute the result $u = c + h + l = x\,y \mod p$ with $x, y, u \in \mathbb{Z}/p\mathbb{Z}$.

Now we consider the question of how to calculate $[l, h, c]$ quickly. Because of the special structure of r, where only two bits are 1, we can use some shift operations to reduce the bit complexity and save on the cost of divisions. Different r's have different non-zero bits, but for clarity of presentation we use a particular radix r, namely $r = 2^{63} + 2^{34}$, for the prime $P = r^k + 1$ with $k = 8$.

Let x_i, y_j be any two digits in the radix r representation of $x, y \in \mathbb{Z}/p\mathbb{Z}$. Since $0 \le x_i, y_j \le r$ holds, we have

$$\begin{aligned} x_i\,y_j &= (x_{i0} + x_i 1\,r)\,(y_{j0} + y_j 1\,r) \\ &= x_{i0}\,y_{j0} + (x_{i0}\,y_{j1} + x_{i1}\,y_{j0})\,r + x_{i1}\,y_{j1}\,r^2 \end{aligned}$$

where $0 \le x_{i0}, y_{j0} < r$, and $x_{i1}, y_{j1} \in \{0, 1\}$. Hence, we have $0 \le x_{i0}y_{j1},\ x_{i1}y_{j0},\ x_{i1}\,y_{j1} < r$. We only need to consider the case of $x_{i0}\,y_{j0}$, where $0 \le x_{i0}\,y_{j0} < r^2 < 2^{127}$. We can rewrite

$$\begin{aligned} x_{i0}y_{j0} &= (a_0 + a_1 2^{32})(b_0 + b_1 2^{32}) \\ &= a_0b_0 + a_0b_1 2^{32} + a_1b_0 2^{32} + a_1b_1 2^{64} \\ &= c_0 + c_1 2^{64}. \end{aligned}$$

Notice that a_0, a_1, b_0, b_1 are in $[0, 2^{32})$, using addition and shift operation, we can rewrite $x_{i0}y_{j0}$ into the form $c_0 + c_1 2^{64}$, where $c_0 < 2^{64}$ and $c_1 < 2^{63}$. Then, we have:

$$\begin{aligned} x_{i0}y_{j0} &= c_0 + c_1 2^{64} \\ &= c_0 + c_1' 2^{63} && \text{where } c_1' = 2\,c_1, 0 \le c_1' < 2^{64} \\ &= c_0 + c_1'(2^{63} + 2^{34}) - c_1' 2^{34} \\ &= c_0 + c_1' r - c_1' 2^{34}, \end{aligned}$$

where the part $c_0 + c_1' r$ can be rewritten into the form of $l + hr + cr^2$ easily.

For $c_1' 2^{34}$, where $0 \le c_1' < 2^{64}$ holds, we observe:

$$\begin{aligned} c_1' 2^{34} &= (d_0 + d_1 2^{29}) 2^{34} && \text{with } 0 \le d_0 < 2^{29}, 0 \le d_1 < 2^{35} \\ &= d_0 2^{34} + d_1 2^{63} \\ &= d_0 2^{34} + d_1(2^{63} + 2^{34}) - d_1 2^{34} \\ &= (d_0 - d_1) 2^{34} + d_1 r \\ &= (e_0 + e_1 2^{29}) 2^{34} + d_1 r && \text{with } |e_0| < 2^{29}, |e_1| < 2^6 \\ &= (e_0 - e_1) 2^{34} + e_1 r + d_1 r. \end{aligned}$$

Since $|(e_0 - e_1)2^{34}| < r$ holds, the number $c_1' 2^{34}$ can easily be rewritten into the form of $l + hr + cr^2$. We add the (l, h, c)-representations of each part together, with some normalization we can get the result we need where $x_i\,y_j = l + h\,r + c\,r^2$.

To summarize, the algorithm below uses only addition and shift operation to compute the (l, h, c)-representation of $x_i y_j$. for $0 \le x_i y_j < 2^{64}$, and $0 \le l, h < r$, and $c \in \{0, 1\}$.

Algorithm 4.1 An algorithm for rewriting $x_i y_j$ into $l + hr + cr^2$

procedure Rewrite($[l, h, c] = [x_i, y_j]$)
  if $x_i \geq r$ **then**
    $x_{i1} := 1$
    $x_{i0} := x_i - r$
  else
    $x_{i1} := 0$
    $x_{i0} := x_i$
  end if ▷ $x_i = x_{i0} + x_{i1} r$
  if $y_i \geq r$ **then**
    $y_{i1} := 1$
    $y_{i0} := y_i - r$
  else
    $y_{i1} := 0$
    $y_{i0} := y_i$
  end if ▷ $y_i = y_{i0} + y_{i1} r$

  $[v_1, v_2, v_3] := [0, x_{i0}\, y_{i1}, 0]$; ▷ $x_{i0} y_{i1} r$
  $[v_4, v_5, v_6] := [0, x_{i1}\, y_{i0}, 0]$; ▷ $x_{i1} y_{i0} r$
  $[v_7, v_8, v_9] := [0, 0, x_{i1}\, y_{i1}]$; ▷ $x_{i1} y_{i1} r^2$

  $c_0 := x_{i0}\, y_{i1} - 2^{64}$
  $c_1 := (x_{i0}\, y_{i1}) >> 64$
  $c_1' := 2\, c_1$ ▷ $x_{i0} y_{i0} = c_0 + c_1 2^{64} = c_0 + c_1'\, 2^{63}$
  if $c_0 \geq r$ **then**
    $[v_{10}, v_{11}, v_{12}] := [c_0 - r, 1, 0]$
  else
    $[v_{10}, v_{11}, v_{12}] := [c_0, 0, 0]$
  end if ▷ $c_0 = v_{10} + v_{11} r + v_{12} r^2$
  if $c_1' \geq r$ **then**
    $[v_{13}, v_{14}, v_{15}] := [0, c_1' - r, 1]$
  else
    $[v_{13}, v_{14}, v_{15}] := [0, c_1', 0]$
  end if ▷ $c_1'\, r = v_{13} + v_{14} r + v_{15} r^2$;

  $d_1 := c_1' >> 29$;
  $d_0 := c_1' - d_1 << 29$;
  $e_1 := (d_0 - d_1) >> 29$;
  $e_0 := (d_0 - d_1 - e_1 << 29)$;
  $[v_{16}, v_{17}, v_{18}] := [(e_0 - e_1) << 34, e_1 + d_1, 0]$;

  $[l, h, c] := [v_1 + v_4 + \cdots + v_{16}, v_2 + v_5 + \cdots + v_{17}, v_3 + v_6 + \cdots + v_{18}]$;
  return $[l, h, c]$;
end procedure

The following algorithm is to calculate $u = x\,y \mod p$.

Algorithm 4.2 Computing $u = x\,y \in \mathbb{Z}/p\mathbb{Z}$ for $x, y \in \mathbb{Z}/p\mathbb{Z}$ using polynomial multiplication

procedure POLYNOMIALMULTIPLICATION($\vec{x}, \vec{y}, r, k$)
  Multiply $f_u(R) = f_x(R) \cdot f_y(R) \mod (R^k + 1)$
  for m from 0 to k - 1 **do**
    $[l_i, h_i, c_i] = \sum_{0 \le i,j<k}^{i+j=m} x_i\,y_j - \sum_{0 \le i,j<k}^{i+j=k+m} x_i\,y_j$
  end for
  ▷ The above k clauses can be executed in parallel

  $ShiftToRight[c_0, c_1, \ldots, c_{k-1}]$
  $ShiftToRight[c_{k-1}, c_0, \ldots, c_{k-2}]$
  $ShiftToRight[h_0, h_1, \ldots, h_{k-1}]$
  $u = c + h + l \mod p$
  return u
end procedure

4.1.2 Based on integer multiplication

This approach is more straight forward. For two numbers x and y in our radix r representation, we map the vectors $\vec{x}$ and $\vec{y}$ to two polynomials $f_x, f_y \in \mathbb{Z}[R]$. Then we evaluate the two polynomials at r, which gives us two integers X and Y, using integer multiplication and modulo operation gives the result $U = X\,Y \mod p$. At last, we only need to convert the product back to the radix r representation. See Algorithm 4.3.

Algorithm 4.3 Computing $x\,y \in \mathbb{Z}/p\mathbb{Z}$ for $x, y \in \mathbb{Z}/p\mathbb{Z}$ using integer multiplication

procedure INTEGERMULTIPLICATION($\vec{x}, \vec{y}, r, k, p$)
  $X := 0\ Y := 0$ ▷ X and Y are GMP integers
  for i from $k - 1$ to 0 **do**
    $X := X \cdot r + x_i$
    $Y := Y \cdot r + y_i$
  end for
  $U := (X \cdot Y) \mod p$
  return GeneralizedFermatPrimeField(U)
end procedure

4.2 Analysis

Here we want to analyze the complexity of multiplication in $\mathbb{Z}/p\mathbb{Z}$, for $p = r^k + 1$, with radix r representation. Since any number in our representation multiplied by any power of r is just a cyclic shift, we now only consider the case that multiplication is between two arbitrary numbers, where both of them are not powers of r.

In the following analysis, we compute $u = x \cdot y$, where $x = x_{k-1}r^{k-1} + \cdots + x_0$ and $y = y_{k-1}r^{k-1} + \cdots + y_0$ are two numbers in our Generalized Fermat Prime Field, with radix r representation. Let M be a multiplication time and let ω be the number of bits in a machine word. We want to analyze the complexity of multiplication with different approaches.

4.2.1 Based on polynomial multiplication

We view x and y as polynomials f_x and f_y in a variable R with integer coefficients $x_0, \ldots, x_{k-1}$ and $y_0, \ldots y_{k-1}$, whose bit sizes are at most that of one machine word. First step in our multiplication is to multiply f_x and f_y in $\mathbb{Z}[R]$, obtaining $f_u = u_{2k-2}R^{2k-2} + \cdots + u_0$. The multiplication time of multiplying two polynomials of degree less than k is $\mathsf{M}(k)$. The complexity of multiplying each pair of coefficients is $\mathsf{M}(\omega)$ and the largest bit size of the coefficients of f_u is $\omega + k$, so the maximum complexity of each operation in the polynomial multiplication is $\max(\mathsf{M}(\omega), \Theta(\omega + k))$, which gives us the total complexity of this step:

$$\mathsf{M}(k) \max(\mathsf{M}(\omega), \Theta(\omega + k)) \tag{4.1}$$

In the next step, we compute the remainder of f_u w.r.t $R^k + 1$. We should notice that computing the remainder here is the same as computing $f_u \mod (R^k + 1)$ that is using -1 to replace every R^k. So, for each term in f_u, if the degree is greater than $k - 1$, reduce the degree by k and reverse the sign for the coefficient. Combining the terms with the same degree gives the final result of this step, $f_u = f_x f_y \mod (R^k + 1) = u_{k-1}R^{k-1} + \cdots + u_0$. The total number of operations that we need to compute the remainder is in the order of $\Theta(k)$, the bit complexity of each operation is $\Theta(\omega + k)$, thus the complexity of this step is:

$$\Theta(k\,\omega) \tag{4.2}$$

Next, we want to write each u_i as $l_i + h_i r + c_i r^2$ with $0 \leq l_i, h_i, c_i < r$ using two divisions (one by r^2 and one by r), we get three vectors $[l_0, \ldots l_{k-1}]$, $[h_0, \ldots h_{k-1}]$ and $[c_0, \ldots c_{k-1}]$. Using cyclic shift on the three vectors, we obtain three numbers in radix r format: z_l, z_h, z_c. We need $2\,k$ divisions in machine word size and three cyclic shifts for this step in total. So the complexity is:

$$\Theta(k\,\mathsf{M}(\omega)) \tag{4.3}$$

The last step in this approach is to add three numbers, z_l, z_h, z_c, together using two additions in $\mathbb{Z}/p\mathbb{Z}$. The complexity is:

$$\Theta(k\,\omega) \tag{4.4}$$

We can see that the second step has the greatest complexity 4.2. Thus, the total complexity of the approach based on polynomial multiplication is in the order of:

$$\Theta(\mathsf{M}(k)) \max(\mathsf{M}(\omega), \Theta(\omega + k)) \tag{4.5}$$

4.2.2 Based on reduction to integer multiplication

In this approach, we convert two numbers in our radix r representation x and y into two big integers X and Y. Then we multiply them together as integers and convert the product to radix-r representation. All of the operations we use in this method can be performed with the GMP library [18].

The GMP library chops the numbers into several parts which are called "limbs". For numbers with different numbers of limbs, GMP uses different multiplication algorithms. Let us consider the case of multiplication between two equal size numbers with N limbs each. For the base case with no threshold, the naive long multiplication is used with complexity of $O(N^2)$. With the minimum of 10 limbs, GMP uses Karatsuba's algorithm with complexity of $O(N^{\log 3/\log 2})$. Furthermore, multi-way Toom multiplication algorithms are introduced. Toom-3 is asymptotically $O(N^{\log 5/\log 3})$, representing 5 recursive multiplies of $1/3$ original size each while Toom-4 has the complexity of $O(N^{\log 7/\log 4})$. Though there seems an improvement over Karatsuba, Toom does more evaluation and interpolation so it will only show its advantage above a certain size. For higher degree Toom 'n' half is used. Current GMP uses both Toom-6 'n' half and Toom-8 'n' half. At large to very large sizes, GMP uses a Fermat style FFT multiplication, following Schönhage and Strassen. Here k is a parameter that controls the split, with FFT-k splitting the number into 2^k pieces, leading the complexity to $O(N^{k/(k-2)})$. It means $k = 7$ is the first FFT that is faster than Toom-3. Practically, the threshold for FFT in the GMP library is found in the range of $k = 8$, somewhere between 3000 and 10000 limbs(See more in GMP library [18] manual).

Firstly, we reduce x and y to X and Y using the following method.

$$X = (((x_{k-1} * r) + x_{k-2}) * r \cdots + x_1) * r + x_0 \tag{4.6}$$

which needs $k-1$ additions and $k-1$ multiplications with at most $k\omega$ bits. Here, we still use M to represent the multiplication time. So, the complexity of this step is:

$$\Theta(k\,\mathsf{M}(k\omega)) \tag{4.7}$$

Then we multiply X and Y using operation from the GMP library. Let $U = X \cdot Y$. The complexity is

$$\mathsf{M}(k\omega) \tag{4.8}$$

At last, U writes $u = u_{k-1}r^{k-1} + \cdots + u_0$ using $k-1$ divisions (by $r^{k-1}, \ldots, r$). The complexity is:

$$\Theta(k\,\mathsf{M}(k\omega)) \tag{4.9}$$

The total complexity of this approach is

$$\Theta(k\,\mathsf{M}(k\omega)) \tag{4.10}$$

4.3 Implementation with C code

In this section we give some details of how we actually implement the multiplication between two arbitrary elements in $\mathbb{Z}/p\mathbb{Z}$. We follow the basic idea of algorithm 4.2 but there are more problems we need to solve.

Let $f_x(R), f_y(R)$ represent $x, y \in \mathbb{Z}/p\mathbb{Z}$ respectively. In the first step of the multiplication, we need to compute $f_u(R) = f_x(R) \cdot f_y(R) \mod (R^k + 1)$ in $\mathbb{Z}/p\mathbb{Z}$, which is a **Negacyclic convolution**. In Section 2.4, we introduced a fast algorithm to compute convolution, which is computing $f(x) \cdot g(x) \mod (x^n - 1)$ for two polynomials f and g with degree less than n. A similar approach can be used for computing the negacyclic convolution.

Let q be a prime, ω be an n-th primitive root of unity in $\mathbb{Z}/q\mathbb{Z}$, and θ be a $2n$-th primitive root of unity in $\mathbb{Z}/q\mathbb{Z}$. Also we have two polynomials $f(x)$ and $g(x)$ with degree less than n, we use $\vec{a}$ and $\vec{b}$ to represent the coefficient vector of the f and g. First, we need to compute two vectors

$$\vec{A} = (1, \theta, \ldots, \theta^{n-1}) \tag{4.11}$$

and

$$\vec{A'} = (1, \theta^{-1}, \ldots, \theta^{1-n}) \tag{4.12}$$

The negacyclic convolution of f and g can be compute as follow

$$\vec{A'} \cdot \mathsf{InverseDFT}(\mathsf{DFT}(\vec{A} \cdot \vec{a}) \cdot \mathsf{DFT}(\vec{A} \cdot \vec{b})) \tag{4.13}$$

All the dot multiplication between vectors are point-wise multiplication. The InverseDFT and DFTs are all n-point. We use unrolled inline DFTs in the implementation. The details of the DFTs are given in Chapter 5. This equation gives the following algorithm.

Algorithm 4.4 is to compute $f_x(R) \cdot f_y(R) \mod (R^k + 1)$ over a finite field $\mathbb{Z}/q\mathbb{Z}$ with q being a machine word size prime and $f_x(R)$, $f_y(R)$ being two polynomials of degree $k - 1$. $\vec{x}$ and $\vec{y}$ are the coefficient lists of f_x and f_y.

Algorithm 4.4 Computing $f_x(R) \cdot f_y(R) \mod (R^k + 1)$ in $\mathbb{Z}/q\mathbb{Z}$ using Negacyclic Convolution

input:
- a prime number q and k is a power of 2 with $k|(q-1)$,
- two vectors $\vec{x}$ and $\vec{y}$ of k elements,contain the coefficients of polynomials $f_x(R)$ and $f_y(R)$.

output:
- a vector $\vec{u}$ that contains the coefficients of polynomial $f_u(R) = f_x(R) \cdot f_y(R) \mod (R^k + 1)$

procedure NEGACYCLICCONVOLUTION($\vec{x}$, $\vec{y}$, q, k)
    ω := PrimitiveRootOfUnity(q, k); ▷ ω is the kth primitive root of unity of q
    θ := PrimitiveRootOfUnity(q, 2k); ▷ θ is the 2kth primitive root of unity of q
    for $0 \le i \le k-1$ **do**
        $A_i := \theta^i \mod q$;
        $x_i := x_i \cdot A_i \mod q$;
        $y_i := y_i \cdot A_i \mod q$;
    end for

    $\vec{x} := \mathsf{DFT}(\vec{x}, \omega, q, k)$;
    $\vec{y} := \mathsf{DFT}(\vec{y}, \omega, q, k)$;

    for $0 \le i \le k-1$ **do**
        $u_i := x_i \cdot y_i \mod q$;
    end for
    $\vec{u} := \mathsf{DFT}(\vec{u}, \omega^{-1} \mod q, q, k)$
    for $0 \le i \le k-1$ **do**
        $A'_i := \theta^{-i} \mod q$;
        $u_i := \frac{1}{k}(u_i \cdot A'_i) \mod q$;
    end for
    return $\vec{u}$
end procedure

Notice that for f_x and f_y in our Generalized Fermat Prime Field $\mathbb{Z}/p\mathbb{Z}$, each coefficient is at most 63 bits. When computing $f_u(R) = f_x(R) \cdot f_y(R) \mod (R^k + 1)$, the size of the coefficients of f_u can be at most $\log k + (2 \cdot 63) = 126 + \log k$, which is more than one machine word, so that we cannot do the computation using single-precision arithmetic. But, multi-precision arithmetic can be very expensive and would make the algorithm inefficient. So we use two machine word negacyclic convolution in stead of one using big numbers. Hence, we need to apply the Chinese Remainder Theorem (CRT) to get the result that we want.

Let p_1 and p_2 be two machine word size prime numbers, so that we have $\mathsf{GCD}(p_1, p_2) = 1$. Then we use the extended Euclidean division to get m_1 and m_2 that satisfy the following relation

$$p_1 m_1 + p_2 m_2 = 1 \tag{4.14}$$

Let a be an integer and we have

$$a_1 \equiv a \mod p_1 \tag{4.15}$$
$$a_2 \equiv a \mod p_2 \tag{4.16}$$

Then we compute $a \mod (p_1\, p_2)$ by

$$a \equiv a_2\, p_1\, m_1 + a_1\, p_2\, m_2 \mod (p_1\, p_2) \tag{4.17}$$
$$= ((a_2\, m_1) \mod p_2)\, p_1 + ((a_1\, m_2) \mod p_1)\, p_2 \tag{4.18}$$

Hence, for $x, y \in \mathbb{Z}/p\mathbb{Z}$, we compute $u_1 = x \cdot y \mod p_1$ and $u_2 = x \cdot y \mod p_2$, then use 4.18 to compute $u = x \cdot y \mod (p_1\, p_2)$. With some normalization we will get $u = x \cdot y \in \mathbb{Z}$. Let $R = k\, r^2$ be the upper bound of $(|u_0|, \ldots, |u_{k-1}|) \in \mathbb{Z}$. To get the correct answer, we need the following restrictions:

1. $R \leq \frac{p_1\, p_2 - 1}{2}$
2. the results we get from the CRT should be normalized so that they fall into the range of $[-\frac{p_1 p_2 - 1}{2}, \frac{p_1 p_2 - 1}{2}]$

If $\frac{p_1\, p_2 - 1}{2} < R$, any result that is in the range of $(\frac{p_1\, p_2 - 1}{2}, R)$ and $(-R, -\frac{p_1\, p_2 - 1}{2})$ will be inaccurate since the modular operation will make it in the range of $[-\frac{p_1\, p_2 - 1}{2}, \frac{p_1\, p_2 - 1}{2}]$.

As we mentioned before, all the results are in the range of $(-R, R)$ in $\mathbb{Z}$, which means $-\frac{p_1 p_2 - 1}{2} < u_i < \frac{p_1 p_2 - 1}{2}$ hold. Hence, after all the normalization we will have all the results in $\mathbb{Z}$ without losing any accuracy.

The small primes p_1 and p_2 are hard coded into the algorithm for now, where both $p_1 = 4179340454199820289$ and $p_2 = 24859869943085137 93$ are 61-bit numbers. So, when choosing the Generalized Fermat prime, we should be very careful because of the two restrictions. For these two primes p_1 and p_2, the size of the chosen Generalized Fermat prime number $p = r^k + 1$ should be as follows:

$$\log \frac{p_1 p_2 - 1}{2} > \log(k\, r^2) \tag{4.19}$$
$$121 > \log k + 2 \log r \tag{4.20}$$
$$\log r < 59 \;\; when \;\; k = 8 \tag{4.21}$$
$$\log r < 58 \;\; when \;\; k = 16 \tag{4.22}$$
$$\log r < 58 \;\; when \;\; k = 32 \tag{4.23}$$
$$\log r < 57 \;\; when \;\; k = 64 \tag{4.24}$$

As we know, the modular operation in 4.18 is expensive, so in the implementation we use what is called **reciprocal division** to reduce the cost of the modular operations.

Let's say we want to compute $a \mod n$, instead of doing one single modular operation, we pre-compute the value of $ninv = 1/n$. Then we compute the result by

$$a - n \cdot a \cdot ninv \equiv a \mod n \tag{4.25}$$

Here, we only keep the integer part of $a \cdot ninv$, so that $n \cdot a \cdot ninv$ gives the quotient of the Euclidean division of a by n.

The following C code give the function of an efficient modular operation using the recisprocal division method.

```
void u64_mod_u64(usfixn64 &a, const usfixn64 &n){
        //a = a % n;
        double ninv = 1 / (double) n;
        usfixn64 q = (usfixn64) ((((double) a)) * ninv);
        usfixn64 res;
        res = a - q * n;
        a = res & (U64_MASK);
}
```

Listing 4.1: Modular funtion using reciprocal division

Unlike modular operation, multiplication between two machine word size number sometimes can cause overflow, but using multi-precision numbers such as the ones given in the GMP library [18] decreases the efficiency. To avoid that, we use two 64-bit numbers to represent the result of multiplication since the size of the result will be at most 128 bits. Let's say the sizes of a and b are at most 64 bits, we compute the multiplication between a and b by

$$s = a \cdot b = s_1 \cdot 2^{64} + s_0 \tag{4.26}$$

where both of s_1 and s_0 are less than 2^{64}.

To make the process even more efficient, we use assembly language in the following function.

```
void __inline__ mult_u64_u64(const usfixn64 & a, const usfixn64 & b,
usfixn64& s0, usfixn64 &s1){
        //      __int128 mult = (__int128) a * (__int128) b;
        //      s0 = mult & (U64_MASK);
        //      s1 = mult >> 64;

        __asm__ (
                "movq  %2, %%rax;\n\t"          // rax = a
                "mulq %3;\n\t"// rdx:rax = a * b
                "movq  %%rax, %0;\n\t"// s0 = rax
                "movq  %%rdx, %1;\n\t"// s1 = rdx
                : "=rm" (s0),"=rm"(s1)
                : "rm"(a), "rm"(b)
                : "%rax", "%rdx");
}
```

Listing 4.2: Multiplication between two 64-bit numbers

We use function 4.2 to compute the $[t_0, t_1] = a_1 m_2$ in equation 4.18. Then we need to do the modular by p_1. We can use a similar method as function 4.1, but all the numbers will be in the size of 128 bits, so we use the representation of $s_1 2^{64} + s_0$.

To keep $1/p_1$ in the correct precision, we multiply it by 2^{128}, and then we get

$$\frac{2^{128}}{p_1} = p_1\_q\, 2^{64} + p_1\_m \tag{4.27}$$

We have a function mult_u128_u128_hi128(see Appendix A, function A.1) to multiply $[t_0, t_1]$ and $[p_1\_q, p_1\_m]$ keeping the higher 64 bits only, which give the quotient q_0 of $a_1 m_2$

divided by p_1. Then we have

$$a_1 m_2 \mod p_1 = a_1 m_2 - q_0 p_1 \tag{4.28}$$

Again we use function 4.2 to get the result of $(a_1 m_2 \mod p_1) \cdot p_2$. Then use the same process to compute $(a_2 m_1 \mod p_2) \cdot p_1$. Adding the two parts together gives us the final result of equation 4.18.

Using the same notation as above, the following algorithm (by Mr. Davood Mohajerani) computes equation 4.18 without using any multi-precision number. The corresponding C code can be found in Appendix A.

Algorithm 4.5 Chinese Remainder Algorithm computing equation 4.18

1: **input:**
   - two machine word size prime numbers $\mathtt{p}_1$ and $\mathtt{p}_2$,
   - $\mathtt{m}_1$ and $\mathtt{m}_2$ such that $p_1 m_1 + p_2 m_2 = 1$ holds,
   - $\mathtt{a}_1$ and $\mathtt{a}_2$ such that $a_1 \equiv a \mod p_1$ and $a_2 \equiv a \mod p_2$ hold.
2: **output:**
   - $a \mod (p_1 p_2) = s_1 2^{64} + s_0$ represented by $[\mathtt{s}_0, \mathtt{s}_1]$.
3: **procedure** CRT($p_1, p_2, m_1, m_2, a_1, a_2$)
4: $[p_1\_q, p_1\_m] := \frac{2^{128}}{p_1}$
5: $[p_2\_q, p_2\_m] := \frac{2^{128}}{p_2}$
6: $[t_0, t_1]$:= multi_u64_u64(a_1, m_2)
7: $[t_2, t_3]$:= multi_u64_u64(a_2, m_1)
8: q_0 := mult_u128_u128_hi128(t_0, t_1, p_1_q, p_1_m)
9: q_1 := mult_u128_u128_hi128(t_2, t_3, p_2_q, p_2_m)
10: $[b_0, b_1]$:= multi_u64_u64(q_0, p_1)
11: $[b_2, b_3]$:= multi_u64_u64(q_1, p_2)
12: $c_1 := [t_0, t_1] - [b_0, b_1]$
13: $c_2 := [t_2, t_3] - [b_2, b_3]$
14: $[s_0, s_1]$:= multi_u64_u64(c_0, p_2) + multi_u64_u64(c_1, p_1)
15: Normalization $[s_0, s_1] \in [-\frac{p_1 p_2 - 1}{2}, \frac{p_1 p_2 - 1}{2}]$
16: **return** $[s_0, s_1]$
17: **end procedure**

After the negacyclic convolutions and the Chinese Remainder algorithm, we have $f_u = f_x \cdot f_y \mod (R^k + 1) \in \mathbb{Z}$. Next, we need to convert the coefficients of f_u into the (l, h, c) representation as we discussed in Section 4.1.1.

Let $u_i = s_1 2^{64} + s_0$ and r be the radix of our Generalized Fermat Prime Field, we use a function div_by_const_R (see Appendix A function A.3) to get $[m_0, q_0], [m_1, q_1]$ and $[m_2, q_2]$ that satisfy the following relation

$$s_0 = q_0 r + m_0 \quad with \quad q_0, m_0 < r \tag{4.29}$$

$$s_1 = q_1 r + m_1 \quad with \quad q_1, m_1 < r \tag{4.30}$$

$$2^{64} = q_2 r + m_2 \quad with \quad q_2, m_2 < r \tag{4.31}$$

Then we compute the $[l, h, c]$ by

$$\begin{aligned}
[l, h, c] &= (q_0\, r + m_0) + (q_1\, r + m_1)\,(q_2\, r + m_2) && (4.32)\\
&= q_1\, q_2\, r^2 + (m_1\, q_2 + m_2\, q_1 + q_0)\, r + (m_0 + m_1\, m_2) && (4.33)\\
&= c'\, r^2 + h'\, r + l' && (4.34)
\end{aligned}$$

Notice that the $[l', h', c']$ we get here is not the final result yet since $h' = m_1\, q_2 + m_2\, q_1 + q_0$ and $l' = m_0 + m_1\, m_2$can be greater than r. We call function div_by_const_R on h' and l' to normalize the result and give us $[l', h', c'] = [l_1, h_1, c_1]r + [l_0, h_0, c_0]$. We use addition with carry to get the final result $[l, h, c] = [l_1, h_1, c_1] + [l_0, h_0, c_0]$.

The following algorithm takes two numbers $[s_0, s_1]$ less than 64 bits as input, and output the $[l, h, c]$ as we defined in Section 4.1.1. The corresponding C code can be found in Appendix A function A.4.

Algorithm 4.6 Computing $s_1\, 2^{64} + s_0 = l + h\, r + c\, r^2$

1: **input:**
   - two machine word size numbers $\mathtt{s_1}$ and $\mathtt{s_0}$,
   - the radix `r`.
2: **output:**
   - $[l, h, c]$ such that $s_1\, 2^{64} + s_0 = l + h\, r + c\, r^2$.
3: **procedure** LHC(s_1, s_0, r)
4: $[q_0, m_0]$:= div_by_const_R(s_0, r)
5: $[q_1, m_1]$:= div_by_const_R(s_1, r)
6: $[q_2, m_2]$:= div_by_const_R(2^{64}, r)
7: $[l', h', c']$:= $(q_0\, r + m_0) + (q_1\, r + m_1)\,(q_2\, r + m_2)$
8: $[l_0, l_1]$:= div_by_const_R(l', r)
9: $[h_0, h_1]$:= div_by_const_R(h', r)
10: $[c_0, c_1]$:= div_by_const_R(c', r)
11: $[l, h, c]$:= $[l_0, h_0, c_0] + [l_1, h_1, c_1]$
12: **return** $[l, h, c]$
13: **end procedure**

Now, we have all the coefficients of f_u in the form of $[l, h, c]$. Rearranging the k $[l, h, c]$ vectors gives us three vectors $\vec{l} = [l_0, \ldots, l_{k-1}], \vec{h} = [h_0, \ldots, h_{k-1}]$ and $\vec{c} = [c_0, \ldots, c_{k-1}]$. Then we use function 3.2 to multiply $\vec{c}$ by r^2 and $\vec{h}$ by r. Finally, we use function 3.1 to add $\vec{l}, \vec{h}, \vec{c}$ together to get the final result of $x\, y \in \mathbb{Z}/p\mathbb{Z}$.

We call the this approach of multiplying two arbitrary elements in $\mathbb{Z}/p\mathbb{Z}$ the FFT-based multiplication in the Generalized Fermat Prime Field (FFT-based multiplication). The complete algorithm is as follow.

Algorithm 4.7 FFT-based multiplication for two arbitrary elements in $\mathbb{Z}/p\mathbb{Z}$

input:
- two vectors $\tilde{\mathbf{x}}$ and $\tilde{\mathbf{y}}$ representing the two elements x and y in $\mathbb{Z}/p\mathbb{Z}$,
- two number r and k such that $p = r^k + 1$ is a generalized Fermat number.

output:
- a vector $\tilde{\mathbf{u}}$ representing the result of $x \cdot y \in \mathbb{Z}/p\mathbb{Z}$.

constant value:
- two machine word size primes p_1 and p_2,
- two numbers m_1 and m_2 such that $p_1 m_1 + p_2 m_2 = 1$ holds.

procedure FFT-basedMultiplication($\vec{x}, \vec{y}, r, k$)
  $\vec{z_1}$:= NegacyclicConvolution($\vec{x}, \vec{y}, p_1, k$)
  $\vec{z_2}$:= NegacyclicConvolution($\vec{x}, \vec{y}, p_2, k$)
  for $0 \le i < k$ **do**
    $[s_{0i}, s_{1i}]$:= CRT($p_1, p_2, m_1, m_2, z_{1i}, z_{2i}$)
  end for
  for $0 \le i < k$ **do**
    $[l_i, h_i, c_i]$:= LHC(s_{0i}, s_{1i}, r)
  end for
  $\vec{c}$:= MulPowR($\vec{c}, 2, k, r$)
  $\vec{h}$:= MulPowR($\vec{h}, 1, k, r$)
  $\vec{u}$:= BigPrimeFieldAddition($\vec{l}, \vec{h}, k, r$)
  $\vec{u}$:= BigPrimeFieldAddition($\vec{u}, \vec{c}, k, r$)
  return $\vec{u}$
end procedure

There are a lot of single-precision modular multiplication in Algorithm 4.7, these modular arithmetic can be very expensive and decrease the efficiency of the whole algorithm, so we decide to use Montgomery multiplication [23] inside this process.

As introduced in Section 2.1.2, Montgomery multiplication requires a special representation of the elements that is for an element $a \in \mathbb{Z}/q\mathbb{Z}$ where q is a machine word size prime, we rewrite a into $(aR \mod q)$ where R is the next power of 2 that is larger than q. In this form, multiplication can be performed efficiently without effect addition and subtraction. The Montgomery multiplication algorithm we use is as follow, supposing the machine word size is 64 bits.

Algorithm 4.8 Montgomery Multiplication in $\mathbb{Z}/q\mathbb{Z}$

1: **input:**
   - two numbers a and b in $\mathbb{Z}/q\mathbb{Z}$,
   - the machine word size prime q,
   - a number $q' = -q^{-1} \mod 2^{64}$
2: **output:**
   - a vector $\tilde{u}$ representing the result of $x \cdot y \in \mathbb{Z}/p\mathbb{Z}$.
3: **constant value:**
   - $c = a\,b\,R^{-1} \mod q$
4: **procedure** MONTGOMERYMULTIPLICATION(a, b, q, q')
5: $R := 2^{64} - 1$
6: $c := a\,b$
7: $d := c\,q'$
8: $c := c + q\,(d\&R)$ ▷ & is the bit-wise and operation
9: $c := c >> 64$ ▷ $>> x$ is shift x bits to the right
10: **if** $c \geq q$ **then**
11: $c := c - q$
12: **end if**
13: **return** c
14: **end procedure**

The C code of the Montgomery multiplication for 64-bit numbers in the BPAS library can be found in Appendix A function A.5 (by Svyatoslav Covanov).

Once we have the Montgomery multiplication function, the "convert-in" and "convert-out" process can be very simple. Let a be an element in $\mathbb{Z}/q\mathbb{Z}$, converting a to the Montgomery representation can be done using the following equation

$$a\,R \equiv \frac{a \cdot R^2}{R} \mod q = \mathsf{MontgomeryMultiplication}(a, R^2, q, q') \tag{4.35}$$

and the converting out from the Montgomery representation can be done by

$$a \equiv \frac{a\,R \cdot 1}{R} \mod q = \mathsf{MontgomeryMultiplication}(a\,R, 1, q, q') \tag{4.36}$$

So far, we have the full implementation of FFT-based multiplication between two arbitrary elements in the Generalized Fermat Prime Field. As we mentioned before, we also have an implementation based on integer multiplication using the GMP library[18] following Algorithm 4.3. The experiment results comparing the two implementations can be found in Chapter 6.

Chapter 5

A generic implementation of FFT over finite fields in the BPAS library

In Section 5.1, we first review the tensor algebra formulation of FFT, following the presentation of [13]. In the same section, we also recall how one can transform the recursive formulation of the six-step DFT to an iterative version, where all DFTs are then performed on a fixed *base-case* size. In the context of Generalized Fermat prime fields, this reduction allows to take advantage of the "cheap" multiplication introduced in Section 3.4. Section 5.2 introduces the different finite fields that are implemented in the *Basic Polynomial Algebra Subprograms*, also known as the BPAS library [3]. For efficiency reasons and convenience purposes, fields with the same functionalities are implemented in both C and C++ languages. In Section 5.3, we explain how we implemented the FFT in the BPAS library following the method in Section 5.1. We show the template functions for different steps in the FFT which can adapt to all the finite fields in the BPAS library. Also, we will explain how we implement the DFT base-cases for 8, 16, 32 and 64 points. This chapter is a joint work with Colin Costello and Davood Mohajerani.

5.1 The tensor algebra formulation of FFT

In the section we review the tensor formulation of FFT. First we define the tensor product of two matrices over a field[24].

Definition 7 *Let n, m, q, s be positive integers and let A, B be two matrices over* `h` *with respective formats $m \times n$ and $q \times s$. The tensor (or Kronecker) product of A by B is an $mq \times ns$ matrix is denoted by $A \otimes B$ and defined by*

$$A \otimes B = \begin{bmatrix} a_{11}B & \cdots & a_{1n}B \\ \vdots & \ddots & \vdots \\ a_{m1}B & \cdots & a_{mn}B \end{bmatrix} \tag{5.1}$$

For example, we have two matrices

$$A = \begin{bmatrix} 0 & 1 \\ 2 & 3 \end{bmatrix} B = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$$

Then we have

$$A \otimes B = \begin{bmatrix} 0 \cdot B & 1 \cdot B \\ 2 \cdot B & 3 \cdot B \end{bmatrix} = \begin{bmatrix} 0 & 0 & 1 & 2 \\ 0 & 0 & 3 & 4 \\ 2 & 4 & 3 & 6 \\ 6 & 8 & 9 & 12 \end{bmatrix}$$

Definition 8 *For matrices A and B, operator $\oplus$ is defined as follow*

$$A \oplus B = \begin{bmatrix} A & 0 \\ 0 & B \end{bmatrix}$$

For n matrices $A_0 \ldots A_{n-1}$, the $\oplus$ sum of them is defined as

$$\bigoplus_{i=0}^{n-1} A_i = A_0 \oplus A_1 \oplus \cdots \oplus A_{n-1} = \begin{bmatrix} A_0 & & & \\ & A_1 & & \\ & & \ddots & \\ & & & A_{n-1} \end{bmatrix}. \tag{5.2}$$

In a ring R, an n-point DFT_n can be seen as a linear map of $R^n \mapsto R^n$. In the BPAS library, we use the six-step recursive FFT algorithm presented in [13]. It can be represented by the following equation

$$\mathsf{DFT}_N = L^N_K (I_J \otimes \mathsf{DFT}_K) L^N_J \, D_{K,J} \, (I_K \otimes \mathsf{DFT}_J) \, L^N_K \;\; \textit{with} \;\; N = J\,K \tag{5.3}$$

which uses the divide-and-conquer idea of Fürer's algorithm. For the part of $I_K \otimes \mathsf{DFT}_J$, we can further expand it to using the base-case DFT_K. Hence, if we have an efficient implementation of the base-case, we will have an efficient algorithm for FFT.

In equation 5.3, L^N_K is called a stride permutation and $D_{K,J}$ is called a twiddle factor. They are defined as follow.

Definition 9 *The stride permutation L^{mn}_m permutes an input vector $\vec{x}$ of length mn as follows*

$$\vec{x}[in + j] \mapsto \vec{x}[jm + i] \tag{5.4}$$

Basically what the stride permutation does is, for an input vector $\vec{x}$ with length mn, it treats the vector as a $n \times m$ matrix and does a transposition on it.

$$L^{mn}_m(M_{n\times m}) = (M_{n\times m})^T \tag{5.5}$$

For example, the input vector is $\vec{x}_8 = [0, 1, 2, 3, 4, 5, 6, 7]$, with $m = 2$ and $nm = 8$, the $n \times m$ matrix is

$$M^T_{n\times m} = \begin{bmatrix} 0 & 1 \\ 2 & 3 \\ 4 & 5 \\ 6 & 7 \end{bmatrix}^T = \begin{bmatrix} 0 & 2 & 4 & 6 \\ 1 & 3 & 5 & 7 \end{bmatrix}$$

So $L^{mn}_m(\vec{x}) = [0, 2, 4, 6, 1, 3, 5, 7]$

Definition 10 *The twiddle factor $D_{K,J}$ is a matrix of the powers of ω.*

$$D_{K,J} = \bigoplus_{j=0}^{K-1} diag\,(1, \omega_i^j, \ldots, \omega_i^{j(J-1)}) \tag{5.6}$$

We can compute all the twiddle factor multiplication with Algorithm 4.7, but as is introduced in Fürer's paper[16], we want to compute the base-case DFT_K using a cheaper multiplication with some K-th primitive roof of unity.

Now, we want to compute DFT_{K^e} by computing DFT_K. The twiddle factor here should be $D_{K,K^{e-s}}$ where $\omega_i = \omega^{K^{s-1}}$ for $(1 \leq s < e)$. And we know from Chapter 3 that for a Generalized Fermat prime $p = r^k + 1$, r is a $2k$-th primitive root of unity, then we have $\omega^N = r^{2k} = 1 \mod p$. Hence, we can using following method to compute the twiddle factor multiplication $y = x \cdot \omega^{i(N/K)+j}$.

$$\begin{aligned} y &= (x \cdot \omega^{iN/K}) \cdot \omega^j && (5.7)\\ &= (x \cdot r^{2ki/K}) \cdot \omega^j && (5.8)\\ &= (x \cdot r^i) \cdot \omega^j && (5.9) \end{aligned}$$

We use Algorithm 3.2 to compute the multiplication with r^i which is very cheap, and only compute the twiddle factor multiplication with ω^j using Algorithm 4.7. We can pre-compute all the power of ω^j for $0 \leq j < N/K$ to further reduce the complexity of the algorithm. In conclusion, to compute DFT on K^e points, we need to pre-compute the power of ω^j for all $0 \leq j < K^{e-1} - 1$.

We can see that once we have an efficient implementation of the base-case DFT_K, we can compute DFT_N at any size where N is some power of 2. In Section 5.3, we will explain how we implement the efficient base-case in the BPAS library.

5.2 Finite fields in the BPAS library

In order to provide both efficiency and convenience, we implemented the following finite fields in the BPAS library using either the C or C++ language.

`SmallPrimeField` **C++ Class:** C++ implementation in the BPAS library of a prime field of the form $\mathsf{GF}(p)$ where p is an arbitrary prime number of machine word size.

`SmallPrimeField` **in C:** Set of C functions in the BPAS library implementing arithmetic operations in a prime field of the form $\mathsf{GF}(p)$ where p is an arbitrary prime number of machine word size.

`BigPrimeField` **C++ Class:** C++ implementation in the BPAS library of a prime field of the form $\mathsf{GF}(p)$ where p is an arbitrary prime number without any restrictions on its size.

`BigPrimeField` **in C:** Set of C functions (provided by the GMP library) implementing arithmetic operations in a prime field of the form $\mathsf{GF}(p)$ where p is an arbitrary prime number without any restrictions on its size.

`GeneralizedFermatPrimeField` **C++ Class:** C++ implementation in the BPAS library of a prime field of the form GF(p) where p is a Generalized Fermat prime, see Chapter 3.

`GeneralizedFermatPrimeField` **in C (GMP-based):** Set of C functions implementing arithmetic operations in a prime field of the form GF(p) where p is a Generalized Fermat prime, see Chapter 3.

`GeneralizedFermatPrimeField` **in C (FFT-based):** Set of C functions implementing arithmetic operations in a prime field of the form GF(p) where p is a Generalized Fermat prime, see Chapter 3. Note that in this case, the multiplication of two elements of the field is done by FFT as we described in 4.3.

Both of the `SmallPrimeField` implementations use machine word size primes (the `long long int` type in C and C++) and have the same functionalities. And all the arithmetic is done using Montgomery representation, see Section 2.1.2. In the C++ class, we convert all the objects into Montgomery representation in the constructor and convert out when users call the convert out method or printing method. The C version has functions for converting in and out, the users should call these functions before and after doing any computations.

Inside the `SmallPrimeField` class, we overload the arithmetic operators $+, -, *, /$ as well as the Boolean operators $==, !=, >, <, >=, <=$; we also have methods for computing the inverse of an elements in the finite field as well as for exponentiation by any integer exponent. For multiplication, we use Algorithm 4.8. Finally, we follow the method introduced in [19] (see Algorithms 2.23 and 2.25) for the Montgomery-based inversion.

The calling sequence of the `SmallPrimeField` class is as follows.

```
#include "bpas.h"
int main(){
        int p = 257;
        SmallPrimeField::setPrime(p);
        //set the prime to 257
        int n = 234;
        SmallPrimeField a(n);
        //create an object that equal to n mod p
        SmallPrimeField b(100);
        //create an object that equal to 100 mod p
        SmallPrimeField a;
        //create a 0 object
        c = a + b;
        c = a - b;
        c = a * b;
        c = a.inverse();
        c = a^5;
        cout << c << endl;
}
```

Listing 5.1: Calling sequence of `SmallPrimeField` class in the BPAS library

An example of using the C implementation of `SmallPrimeField` follows.

```
#include "bpas.h"
int main(){
        long int p = 257;
        long int Pp = getPp(p,R);
        //R can be computed as 2^64 mod p
        long int a = 100;
        long int b = 576;
        a = covert_in(a, p,R);
        b = covert_in(b, p,R);
        a = add(a,b,p);
        //a = a + b mod p
        a = sub(a,b,p);
        //a = a - b mod p
        a = multi(a,b,p,R,Pp);
        //a = a*b/R mod p;
        a = covert_out(a, p,R);
}
```

Listing 5.2: Calling sequence of `SmallPrimeField` macro in the BPAS library

Chapter 6 shows the experimental data of FFT over `SmallPrimeField` in C and C++.

The `BigPrimeField` class has the same functionality as the `SmallPrimeField` class, except that all the arithmetic is done using GMP integers (type mpz_class). So users can choose prime numbers of any size.

The `GeneralizedFermatPrimeField` Class and `GeneralizedFermatPrimeField` C functions follow the representation and arithmetic we introduced in Chapter 3. We implemented multiplication between two arbitrary element using both FFT-based method and GMP-based method in the C version. The default one for overloading the operator $*$ in the class is the GMP-based one.

5.3 BPAS implementation of the FFT

In the BPAS library, we implemented an FFT algorithm using the six-step FFT we described in Section 5.1. Recall the six-step FFT formula

$$\mathsf{DFT}_N = L^N_K\,(I_J \otimes \mathsf{DFT}_K) L^N_J\, D_{K,J}\,(I_K \otimes \mathsf{DFT}_J)\, L^N_K \;\; with \;\; N = J\,K$$

where L is the stride permutation, and D is the twiddle factor multiplication.

Other than the three steps of the permutation and one call of twiddle factor multiplication, we still need to perform the the base-case DFT_K as we explained in Section 5.1. Inside the BPAS library, we implemented base-cases for $K = 8, 16, 32, 64$ and reduced them into DFT_2. First, let us see the function for computing $\mathsf{DFT}_2(x_0, x_1)$

$$\mathsf{DFT}_2(x_0, x_1) = (x_0 + x_1, x_0 - x_1) \tag{5.10}$$

For $K = 2^n$, we reduce DFT_K to DFT_2 by

$$\mathsf{DFT}_{2^n} = L^{2^n}_2\,(I_{2^{n-1}} \otimes \mathsf{DFT}_2)\, L^{2^n}_{2^{n-1}}\, D_{2,2^{n-1}}\,(I_2 \otimes \mathsf{DFT}_{2^{n-1}})\, L^{2^n}_2 \tag{5.11}$$

We follow Algorithm 5.1 to compute a N-point DFTs where $N = K^e$ and e is a positive integer.

Algorithm 5.1 Computing DFT on K^e points in $\mathbb{Z}/p\mathbb{Z}$

input:
- size of the base-case K(8,16,32 or 64), a positive integer e,
- a vector $\vec{x}$ of size K^e,
- ω which is a K^e-th primitive root of unity in $\mathbb{Z}/p\mathbb{Z}$.

output:
- the final result stored in $\vec{x}$

procedure DFT_GENERAL($\vec{x}, K, e, \omega,$)
  for $0 \leq i < e-1$ **do**
    for $0 \leq j < K^i$ **do**
      stride_permutation($\&x_{jK^{e-i}}, K, K^{e-i-1}$)
    end for
  end for ▷ Step 1
  $\omega_a := \omega^{K^{e-1}}$
  for $0 \leq j < K^{e-1}$ **do**
    $idx := jK$
    DFT_K($\&x_{idx}, \omega_a$)
  end for ▷ Step 2
  for $e-2 \geq i \geq 0$ **do**
    $\omega_i := \omega^{K^i}$
    for $0 \leq j < K^i$ **do**
      $idx := jK^{e-i}$
      twiddle($\&x_{idx}, K^{e-i-1}, K, \omega_i$) ▷ Step 3
      stride_permutation($\&x_{idx}, K^{e-i-1}, K$) ▷ Step 4
    end for
    for $0 \leq j < K^{e-1}$ **do**
      $idx := jK$
      DFT_K($\&x_{idx}, \omega_a$)
    end for ▷ Step 5
    for $0 \leq j < K^i$ **do**
      $idx := jK^{e-i}$
      stride_permutation($\&x_{idx}, K, K^{e-i-1}$)
    end for
  end for ▷ Step 6
end procedure

The same code for stride permutation (function stride_permutation in Algorithm 5.1) is used for all BPAS finite fields. Indeed that part is independent of the finite field used for the FFT. The C code of the stride permutation is listed below.

```
1 void stride_permutation(ELEMENTS* A,int m, int n){
2         int blocksize=m^((m^n)&(-(m>n)));
```

```
3           blocksize=BLOCKSIZE^((BLOCKSIZE^blocksize)&(-(BLOCKSIZE>
               blocksize)));
4           ELEMENTS* B = new ELEMENTS[m*n];
5           for (int i = 0; i < n; i += blocksize) {
6                   for (int j = 0; j < m; j += blocksize) {
7                   // transpose the block beginning at [i,j]
8                           for (int k = i; k < i + blocksize; ++k) {
9                                   for (int l = j; l < j + blocksize; ++l)
                                       {
10                                          B[k+l*n] = A[l+k*m];
11                                  }
12                          }
13                  }
14          }
15          for (long int i=0;i<m*n;i++)
16                  A[i]=B[i];
17  }
```

Listing 5.3: Stride permutation for FFT

The same template code for twiddle factor multiplication (function twiddle in Algorithm 5.1) is used for all BPAS finite fields. This template code has 4 specializations

- one for both `SmallPrimeField` (C and C++); switching between C and C++ is done by compilation directive
- one for each of `BigPrimeField` (C and C++);
- one for `GeneralizedFermatPrimeField` (C and C++); switching between C and C++ is done by compilation directive.

The C code of the twiddle template function is as follows. The only difference for different prime fields is the multiplication used in line 5 and 6.

```
1   void twiddle(ELEMENTS* vector, int m, int n, ELEMENTS omega_w){
2           for (int j=0;j<n;j++){
3                   for(int i=0;i<m;i++){
4                           ELEMENTS t;
5                           t=POW(omega_w,(i*j));
6                           vector[j*m+i]=vector[j*m+i]*(t);
7                   }
8           }
9   }
```

Listing 5.4: Twiddle factor multiplication for FFT

For the base-case, that is, DFT_K in Algorithm 5.1, the same template code for is used for all BPAS finite fields. Similarly to the function twiddle, specializations are provided for each BPAS finite field. Three specializations differ by their calls to functions doing addition, subtraction and multiplication. Note that for multiplication by a power of the primitive root, in the case of `GeneralizedFermatPrimeField`, we use the techniques described in Section 3.4

Now, let us consider the base-case of $K = 8$, where ω is an 8-th primitive root in $\mathsf{GF}(p)$.

$$\begin{aligned}
\mathsf{DFT}_8 &= L^8_2 (I_4 \otimes \mathsf{DFT}_2)\, L^8_4\, D_{2,4}\, (I_2 \otimes \mathsf{DFT}_4)\, L^8_2 && (5.12)\\
\mathsf{DFT}_4 &= L^4_2 (I_2 \otimes \mathsf{DFT}_2)\, L^4_2\, D_{2,2}\, (I_2 \otimes \mathsf{DFT}_2)\, L^4_2 && (5.13)\\
\mathsf{DFT}_8 &= L^8_2 (I_4 \otimes \mathsf{DFT}_2)\, L^8_4\, D_{2,4}\, (I_2 \otimes (L^4_2 (I_2 \otimes \mathsf{DFT}_2)\, L^4_2\, D_{2,2}\, (I_2 \otimes \mathsf{DFT}_2)\, L^4_2)\, L^8_2 && (5.14)
\end{aligned}$$

where

$$\begin{aligned}
D_{2,4} &= (1, 1, 1, 1, \omega^0_0, \omega^1_0, \omega^2_0, \omega^3_0) && (5.15)\\
D_{2,2} &= (1, 1, \omega^0_1, \omega^1_1) && (5.16)
\end{aligned}$$

For a prime field with an arbitrary p, we have for DFT_8, $\omega_0 = \omega^{N/K} = \omega$ and for DFT_4, $\omega_1 = \omega^{(N/K)^2} = \omega^2$.

For a Generalized Fermat prime field where the prime is $p = r^4 + 1$ we have for DFT_8, $\omega_0 = \omega^{N/K} = r^{2k/K} = r$ and for DFT_4, $\omega_1 = \omega^{(N/K)^2} = r^2$. Then, the twiddle factors are

$$\begin{aligned}
D_{2,4} &= (1, 1, 1, 1, 1, r, r^2, r^3) && (5.17)\\
D_{2,2} &= (1, 1, 1, r^2) && (5.18)
\end{aligned}$$

Hence, multiplication with the twiddle factors can be done by cyclic shift from Section 3.4.

Now, we follow Equation (5.14) from right to left and get the following unrolled algorithm for DFT_8.

Algorithm 5.2 Unrolled DFT base-case when $K = 8$

```
1:  procedure DFT8(a⃗, ω_i)
2:      DFT2(&a_0, &a_4);
3:      DFT2(&a_2, &a_6);
4:      DFT2(&a_1, &a_5);
5:      DFT2(&a_3, &a_7);                          ▷ dft on permutated indexes
6:
7:      a_6 := a_6 ω^2;
8:      a_7 := a_7 ω^2;                            ▷ twiddle
9:
10:     DFT2(&a_0, &a_2);
11:     DFT2(&a_4, &a_6);
12:     DFT2(&a_1, &a_3);
13:     DFT2(&a_5, &a_7);                          ▷ dft on permutated indexes
14:
15:     a_5 := a_5 ω^1;
16:     a_3 := a_3 ω^2;
17:     a_7 := a_7 ω^2;                            ▷ twiddle
18:
19:     DFT2(&a_0, &a_1);
20:     DFT2(&a_4, &a_5);
21:     DFT2(&a_2, &a_3);
22:     DFT2(&a_6, &a_7);                          ▷ dft on permutated indexes
23:
24:     swap(&a_1, &a_4);
25:     swap(&a_3, &a_6);                          ▷ final permutation
26:     return a⃗;
27: end procedure
```

The swap function swap the value of of its two parameters. The other DFT base-case codes are relatively long so we only show the number of lines here. The numbers of lines for unrolled DFT_K are shown in Table 5.1 (not counting comments). The C code can be found in the BPAS library.

K	8	16	32	64
number of lines	19	55	141	359

Table 5.1: Numbers of lines in n-point unrolled FFT.

Finally, and consequently, the same template code for Algorithm 5.1 is used for all BPAS finite fields.

Chapter 6

Experimentation

In this chapter, we present experimental data of FFT over the finite fields in the BPAS library. In Section 6.1, we compare our implementation of FFT over `SmallPrimeField` Class and C functions as well as another highly optimized FFT implementation from the BPAS library. Also, we compare the two implementations of the multiplication in $\mathbb{Z}/p\mathbb{Z}$ introduced in Chapter 4; the results are in Section 6.2.

In Section 6.3, we report results of FFT over over `BigPrimeField`, `GeneralizedFermatPrimeField` using GMP-based multiplication and `GeneralizedFermatPrimeField` using FFT-based multiplication written in C. Clearly, the latter scenario gives better running tines than the other two.

All the experimental results have been verified using Maple, Python and GMP [18]. This chapter is a joint work with Colin Costello and Davood Mohajerani.

6.1 FFT over small prime fields

Before the work reported in this thesis, various implementations of FFT over small finite fields were developed in the BPAS library. In particular, a highly optimized version by Svyatoslav Covanov is presented in [3]. For this latter, the source code of the FFT is generated at compile time: it takes into account the characteristics of the targeted hardware and it is specialized for a particular prime field. This latter feature allows compiler optimization strategies which are not possible for a generic implementation like the one presented in Chapter 5.

Nevertheless, it is interesting to compare our generic implementation (over the `SmallPrimeField` class and `SmallPrimeField` in C) against the highly optimized FFT produced by Covanov's code generator.

As introduced in Section 5.3, our implementation of FFT is based on an unrolled code for the base-case DFT functions DFT_K, where K can be 8,16,32 or 64. In the following results, we refer to Svyatoslav Covanov's implementation as *Svyatoslav*, and refer to our implementation of FFT using base-case DFT_K as DFT_K_C++ and DFT_K_C depending on which `SmallPrimeField` (C++ class or C functions) it uses.

Figures 6.1, 6.2 and 6.3 show the time spending on FFT over large vectors using base-case size of 8, 16 and 32 respectively. The x-axis gives the size of the vectors. The y-axis is time in seconds. All the results are based on average time of 50 trails. We can see that the C++ class

is slower than the C functions with the implementation of the same algorithm.

Our best result is still slower than Svyatoslav's by the factor of 5. As mentioned above, this is because his code is specialized at the prime number together with embedded assembly code. All the experimental results in this chapter were realized on an Intel(R) Core(TM) 2.90GHz i5-528U CPU.

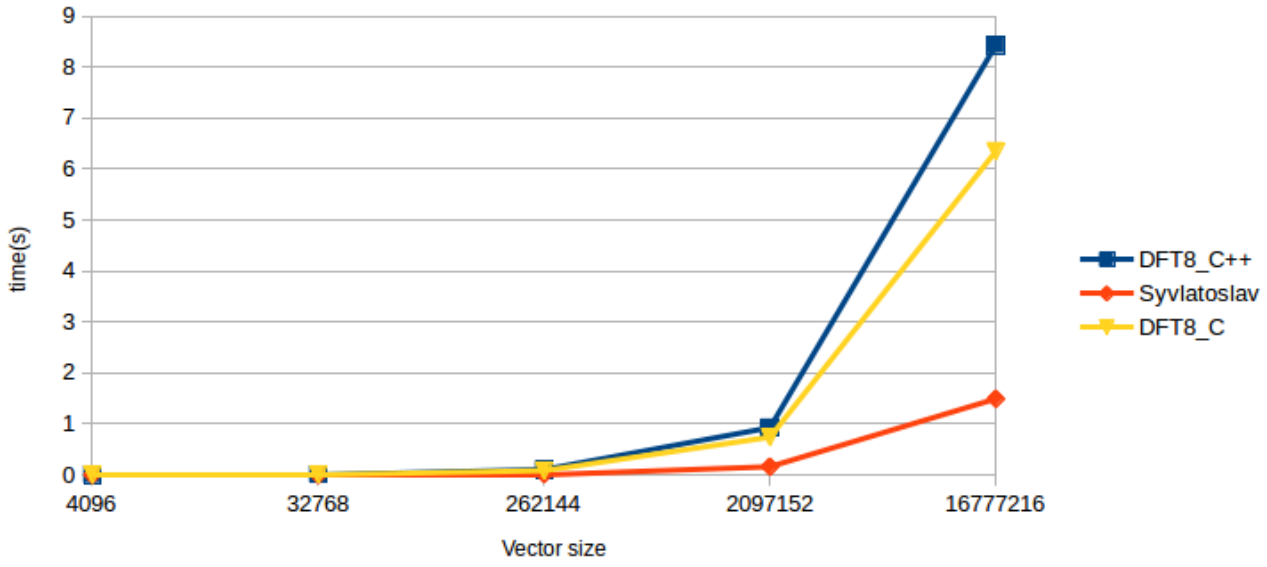

Figure 6.1: FFT over small prime field with DFT_8

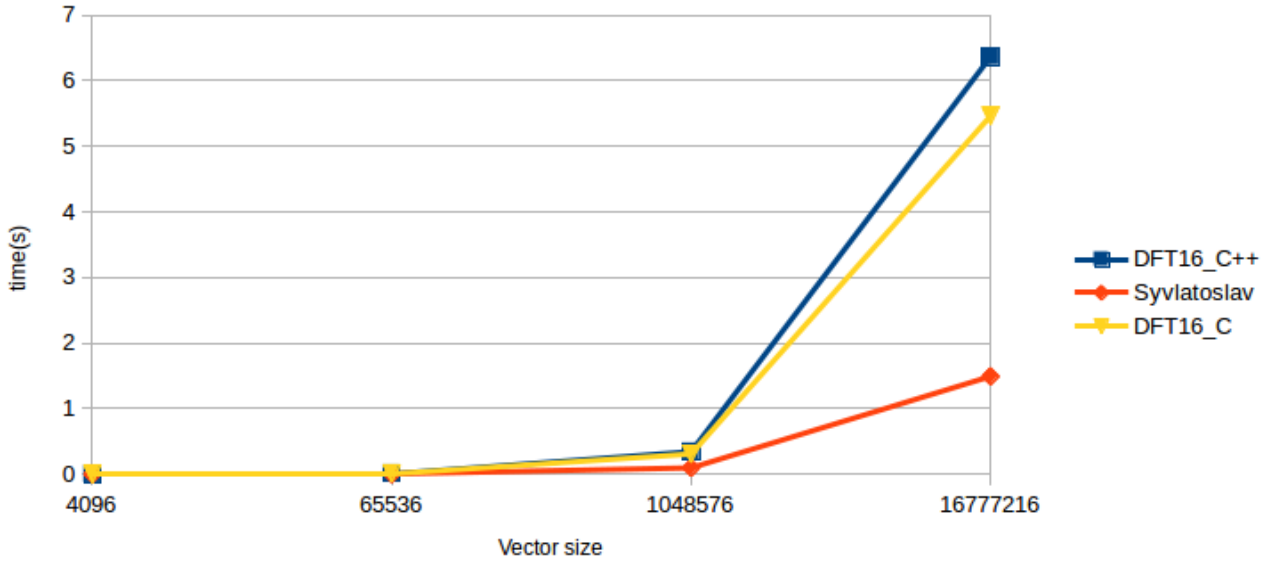

Figure 6.2: FFT over small prime field with DFT_16

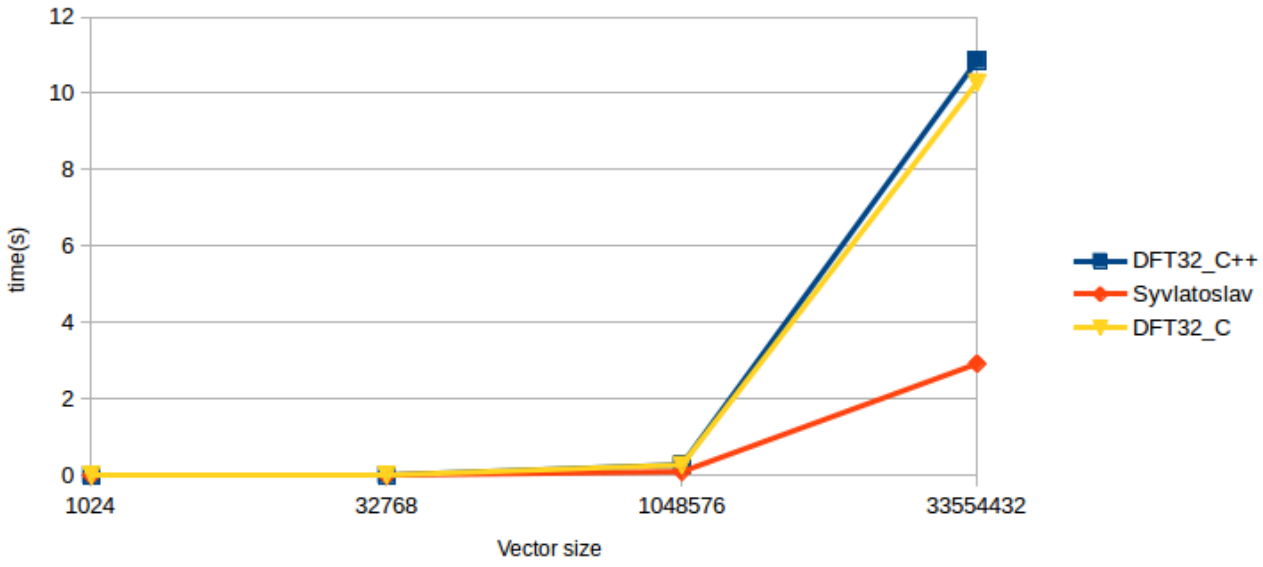

Figure 6.3: FFT over small prime field with $\mathsf{DFT}_3 2$

6.2 Multiplication in generalized Fermat prime fields

As in Chapter 4, we have two multiplication algorithms between two arbitrary elements of the generalized Fermat prime field $\mathbb{Z}/p\mathbb{Z}$. One of them is based on negacyclic convolution using unrolled DFT base-case 4.1.1 (referred to as FFT-based in the figures and tables), the other one is based on GMP integer multiplication 4.1.2 (referred to as GMP-based in the figures and tables). We want to compare the time cost of these two approaches. Also we want to see where we are comparing with big integer modular multiplication using GMP library, where we don't use radix representation of the numbers but use the integer type provided by the GMP library.

We gave the same input to the three multiplication functions, and verified the results against each other. Table 6.1 shows the time costs of one multiplication operation using the three different approaches with regard to k (where $p = r^k + 1$). The time given is in 10^{-6} second scale. We can see clearly that the FFT-based multiplication is faster than the GMP-based one. And the speedup is more obvious when k increases. But both of our approaches are slower than using pure GMP functions.

Figure 6.4 shows the cost ratio of FFT-based and GMP-based multiplication versus GMP multiplication.

k	FFT-based	GMP-based	GMP
8	1303.38	1443.05	224.03
16	2602.56	2886.63	471.45
32	5500.56	6865.14	1282.36
64	10656.10	17649.23	3032.44

Table 6.1: Time cost of one multiplication operation using FFT-based, GMP-based and GMP approaches.

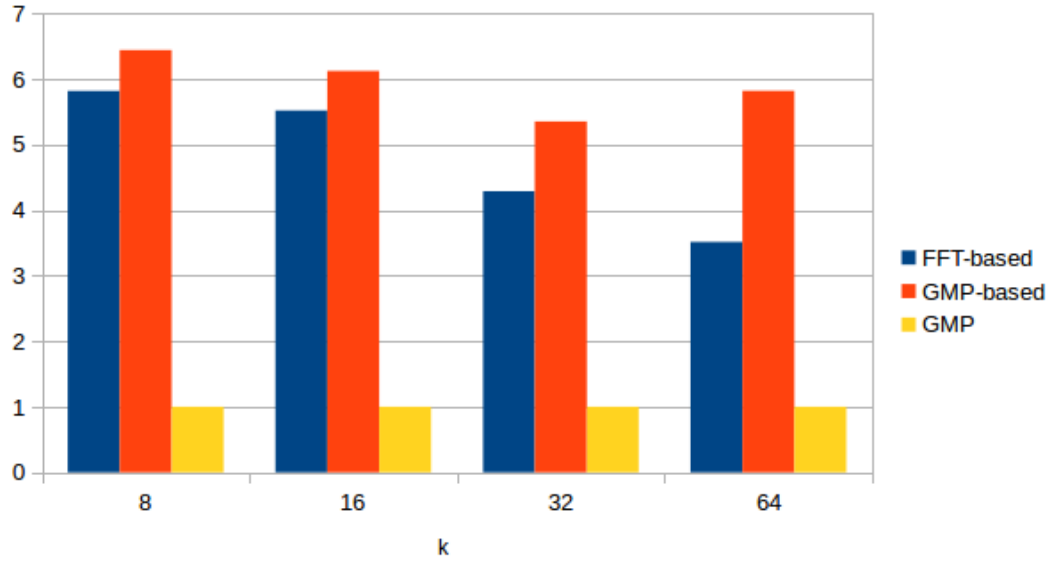

Figure 6.4: FFT-based multiplication vs. GMP-based multiplication vs. GMP multiplication

As introduced in Section 4.3, the FFT-based multiplication (in the generalized Fermat prime field) takes several steps:

Step 1 convert the input elements into Montgomery representation
Step 2 negacyclic convolution
Step 3 convert the result out from Montgomery representation
Step 4 Chinese Remainder Theorem Algorithm
Step 5 LHC algorithm
Step 6 cyclic shift and addition to get the final result.

Figure 6.5 shows the time costs of the above 6 steps *w.r.t* k. Table 6.2 shows the percentage of running time for each step over the total time of the multiplication operation. Convolution takes the dominate part of the cost which fits in the analysis we made in Section 4.2.1

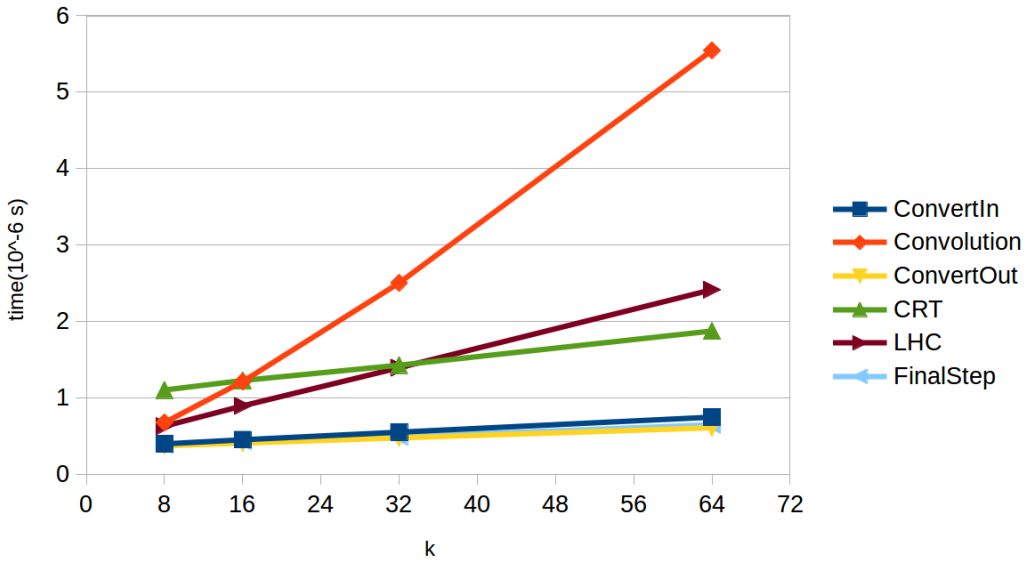

Figure 6.5: Time spends in different parts of the FFT-based multiplication

k	ConvertIn	Convolution	ConvertOut	CRT	LHC	Final step
8	11.17	18.98	10.44	30.88	17.65	10.89
16	9.77	26.20	8.84	26.51	19.34	9.34
32	8.06	36.59	6.97	20.86	20.37	7.16
64	6.32	46.83	5.14	15.83	20.40	5.48

Table 6.2: Time cost in different parts of the FFT-based multiplication in percentage.

6.3 FFT over big prime fields

In this section, we provide experiment data for FFT over big prime fields. The FFT function we use is that of Algorithm 5.1, except that we pre-compute all the power of ω and passed them as input to the algorithm; this is a standard optimization in FFT code over finite fields [24].

We compare FFT computation using the arithmetic over the following finite fields:

- `GeneralizedFermatPrimeField` in C functions (FFT-based)5.2
- `GeneralizedFermatPrimeField` in C functions (GMP-based)5.2
- `BigPrimeField` in C functions (GMP) 5.2

where K is the base-case size and K^e is the input vector size. We should notice that for a prime number $p = r^k + 1$, the base-case size we choose should always satisfy $K = 2k$. Table 6.3 gives the prime numbers we use for different base-cases.

Table 6.4 gives the time cost of FFT on vector with size K^e over the three prime fields. Figure 6.6 shows the cost ratio of GMP-based and GMP versus FFT-based. We can clearly see that FFT over GeneralizedFermatPrimeField using FFT-based multiplication is faster than the other two while BigPrimeField using GMP C functions beats the GMP-based one as the vector size increasing.

K	k	r
16	8	$2^{59} + 2^{16}$
32	16	$2^{58} + 2^{10}$
64	32	$2^{56} + 2^{21}$

Table 6.3: Primes used for different base-cases

K	e	FFT-based	GMP-based	GMP
16	2	0.211	0.281	0.348
16	3	5.961	8.287	8.669
32	2	1.819	2.49	2.47
32	3	109.681	152.877	140.342
64	2	15.775	22.688	22.912
64	3	1995.939	2865.527	2626.658

Table 6.4: Time cost of FFT on vector size K^e over different prime fields

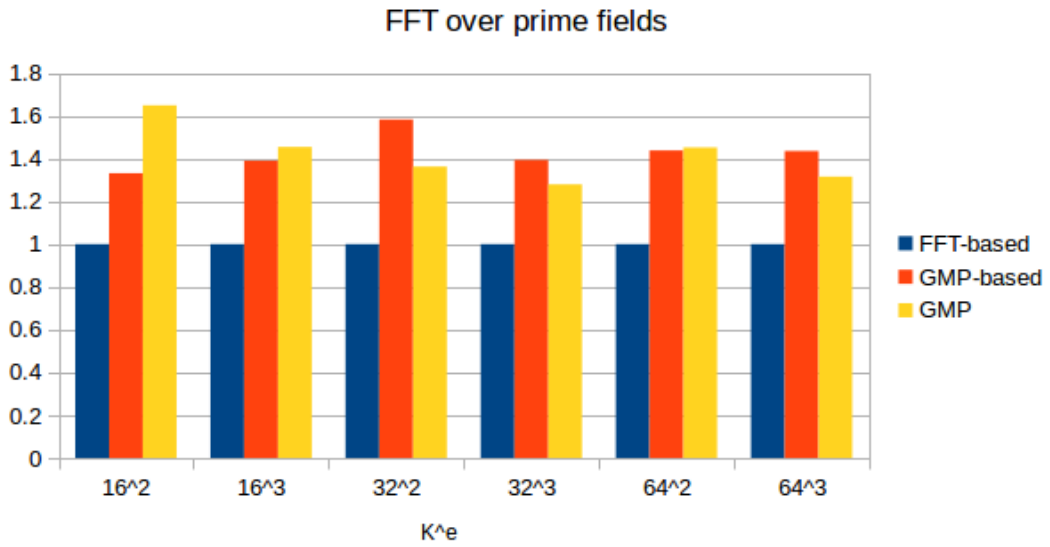

Figure 6.6: FFT of size K^e where $K = 16$

Table 6.5 gives the high-level profiling data on different steps in the FFT algorithm for $K = 64$ and $e = 3$. For both FFT-based and GMP-based implementations of `GeneralizedFermatPrimeField`, most of the time is spent on twiddle factor multiplication where we need to multiply two arbitrary elements in the fields. Comparing with the GMP one, we spent less time in the base-case DFTS, since we only use shift for the multiplication inside the base-case code and that is where we gain our speed up. This profiling result agrees with our original thought of using the trick from Fürer's paper[15] as we explained in Section 1.1.

We can see from Figure 6.4 that for multiplication between two arbitrary elements in a big prime field, the two implementations of ours (FFT-based and GMP-based) are both slower than pure GMP arithmetic. But Figure 6.6 shows that for computing a FFT over big vectors, using GeneralizedFermatPrimeField arithmetic with FFT-based multiplication can be more ef-

time(ms)	permutation	DFT_K	Twiddle
FFT-based	8.08	1400.53	3460.98
GMP-based	7.84	1307.23	6996.69
GMP	721.98	6418.14	1551.41

Table 6.5: Time spend in different parts of the FFT function when $K = 64, e = 3$

ficient than using pure GMP arithmetic. The main reason is that most of the multiplications are done by the cheap (actually linear time) multiplication, see Section 3.4 in the GeneralizedFermatPrimeField while for pure GMP arithmetic all the multiplications are done using the same algorithm.

Table **??** shows the average time spending in one modular multiplication operation in FFT on vectors with size K^e. Figure 6.7 gives the radio of GMP-based and GMP versus FFT-based. We can see that, when computing FFT over Generalized Fermat prime fields, the average time of multiplication operation is less than that of GMP arithmetic. Now we can prove that by using the cheap multiplication with the power of r, we can lower the average time spent in multiplication, and further speed up the FFT process.

K	e	FFT-based	GMP-based	GMP
16	2	0.000179	0.000299	0.00018
16	3	0.000197	0.000287	0.000221
32	2	0.00031	0.000417	0.000389
32	3	0.000354	0.00048	0.000415
64	2	0.000553	0.000816	0.001095
64	3	0.000652	0.000972	0.001157

Table 6.6: Average multiplication time of FFT over big prime fields (Time is in ms)

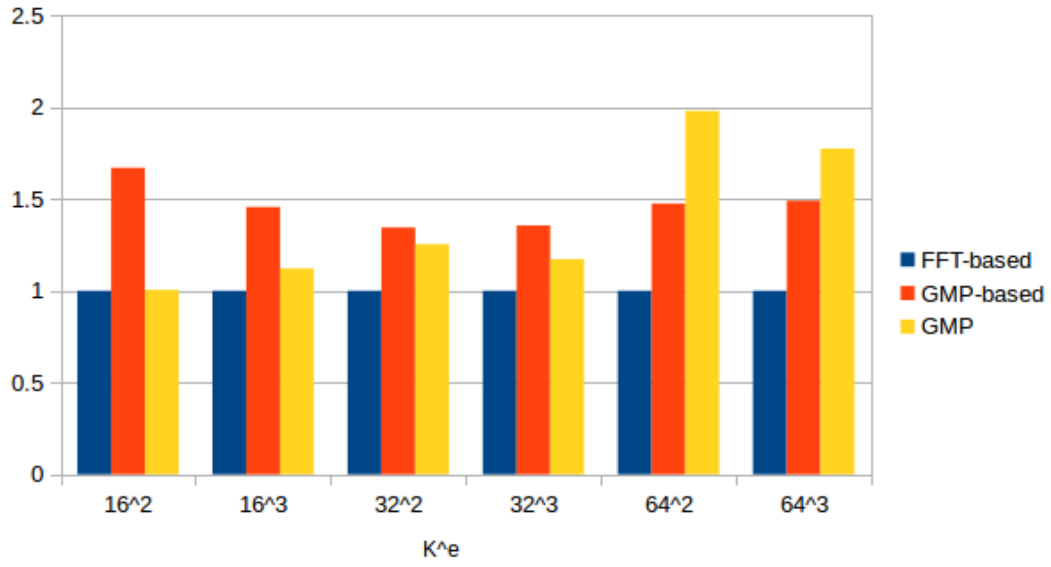

Figure 6.7: Average time of one multiplication operation in FFT

Chapter 7

Conclusion

In thesis, we have proved that FFT can be used effectively to improve the multiplication arithmetic inside big prime fields. Also, we can see that Fürer's trick can lower the average time of the multiplication operations in the FFT of large vectors over such big prime fields.

In Chapter 3 we discussed Generalized Fermat prime numbers which are prime numbers in the form of $p = r^k + 1$, where r is a sparse radix and k is a power of 2. Then we gave algorithms on how to do arithmetic over Generalized Fermat prime fields. Note that multiplying any elements in the fields with some power of r can be done efficiently using a so-called cyclic shift operation. Since r is a $2k$-th primitive root of unity in the prime field, according to Fürer's trick, for a DFT on a vector size of $2k$, all the multiplication can be done via this cyclic shift.

In Chapter 4, we gave the details on how we implement the multiplication between two arbitrary elements in a Generalized Fermat prime field. We gave the algorithm and analysis of the two approaches, FFT-based multiplication and GMP-based multiplication. In the FFT-based one, we used the negacyclic convolution, the Chinese Remainder Theorem and the (l, h, c)-approach to make the algorithm more efficient. And in Chapter 5, we gave a brief overview on the generic FFT implementation in the BPAS library, where we unfold the big DFT down to base-case DFT on $2k$ points.

At last, in Chapter 6, our experimental result shows that, the FFT-based multiplication algorithm is more efficient than the GMP-based one. And by using Fürer's trick, the average time spent on multiplication operations in a FFT process is reduced.

Now we see that, with Fürer's trick, where we use cheap operations to replace some of the multiplication operations, we can improve the performance of FFT of large vectors over big prime fields. Meanwhile, the multiplication between arbitrary elements in a field is still a bottle-neck. So our future work lies in how to improve the multiplication over a big prime field. And whether we can put the whole Fürer's algorithm into practice is still an open question.

Bibliography

[1] E. A. Arnold. Modular algorithms for computing Gröbner bases. *J. Symb. Comput.*, 35(4):403–419, 2003.

[2] C. Chen, R. M. Corless, M. Moreno Maza, P. Yu, and Y. Zhang. An application of regular chain theory to the study of limit cycles. *I. J. Bifurcation and Chaos*, 23(9), 2013.

[3] C. Chen, S. Covanov, F. Mansouri, M. Moreno Maza, N. Xie, and Y. Xie. The basic polynomial algebra subprograms. In *Mathematical Software - ICMS 2014 - 4th International Congress, Seoul, South Korea, August 5-9, 2014. Proceedings*, pages 669–676, 2014.

[4] L. Chen, S. Covanov, D. Mohajerani, and M. Moreno Maza. Big prime field FFT on the GPU. In *Proceedings of the 2017 ACM on International Symposium on Symbolic and Algebraic Computation, ISSAC 2017, Kaiserslautern, Germany, July 25-28, 2017*, pages 85–92, 2017.

[5] The Computational Algebra Group in the School of Mathematics and Statistics at the University of Sydney. *The MAGMA Computational Algebra System for Algebra, Number Theory and Geometry*. http://magma.maths.usyd.edu.au/magma/.

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

[7] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009.

[8] X. Dahan, M. Moreno Maza, É. Schost, W. Wu, and Y. Xie. Lifting techniques for triangular decompositions. In M. Kauers, editor, *ISSAC 2005, Proceedings*, pages 108–115. ACM, 2005.

[9] A. De, P. Kurur, C. Saha, and R. Saptharishi. Fast integer multiplication using modular arithmetic. In *STOC*, pages 499–506, 2008.

[10] A. De, P. P. Kurur, C. Saha, and R. Saptharishi. Fast integer multiplication using modular arithmetic. *SIAM J. Comput.*, 42(2):685–699, 2013.

[11] J. Dumas, T. Gautier, and C. Pernet. Finite field linear algebra subroutines. In *ISSAC 02*, pages 63–74. ACM, 2002.

[12] A. Filatei, X. Li, M. Moreno Maza, and Éric Schost. Implementation techniques for fast polynomial arithmetic in a high-level programming environment. In *Symbolic and Algebraic Computation, International Symposium, ISSAC 2006, Genoa, Italy, July 9-12, 2006, Proceedings*, pages 93–100, 2006.

[13] F. Franchetti and M. Püschel. FFT (fast fourier transform). In *Encyclopedia of Parallel Computing*, pages 658–671. 2011.

[14] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. *ACM Transactions on Algorithms*, 8(1):4, 2012.

[15] M. Fürer. Faster integer multiplication. In *Proceedings of the 39th Annual ACM Symposium on Theory of Computing, San Diego, California, USA, June 11-13, 2007*, pages 57–66, 2007.

[16] M. Fürer. Faster integer multiplication. *SIAM J. Comput.*, 39(3):979–1005, 2009.

[17] J. Gathen and J. Gerhard. *Modern Computer Algebra (3. ed.)*. Cambridge University Press, 2013.

[18] Torbjrn Granlund and the GMP development team. *GNU MP: The GNU Multiple Precision Arithmetic Library*, 5.0.5 edition, 2012. `http://gmplib.org/`.

[19] D. Hankerson, A.J. Menezes, and S. Vanstone. *Guide to Elliptic Curve Cryptography*. Springer Publishing Company, Incorporated, 1st edition, 2010.

[20] http://www.linalg.org/. *LinBox*. The LinBox group, 2005.

[21] A. Karatsuba and Yu. Ofman. Multiplication of multidigit numbers on automata. *Soviet Physics Doklady*, (7):595–596, 1963.

[22] X. Li, M. Moreno Maza, R. Rasheed, and É. Schost. The modpn library: Bringing fast polynomial arithmetic into maple. *J. Symb. Comput.*, 46(7):841–858, 2011.

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

[24] Wei Pan. *Algorithmic Contributions to the Theory of Regular Chains*. PhD thesis, he University of Western Ontario, 2011.

[25] A. Schönhage and V. Strassen. Schnelle multiplikation großer zahlen. *Computing*, 7(3-4):281–292, 1971.

[26] V. Shoup. A new polynomial factorization algorithm and its implementation. *J. Symb. Comp.*, 20(4):363–397, 1995.

[27] W. Strassen. Gaussian elimination is not optimal. *Numerische Mathematik.*, 13:354–356, 1969.

Appendix A

C Functions for Multiplication in Generalized Fermat Prime Field

```
1  //mult_u128_u128_hi128: returns ((x0+x1.u64)*(y0+y1.u64))>>128
2  void __inline__ mult_u128_u128_hi128(const usfixn64 & x0, const usfixn64
       & x1,
3  const usfixn64 & y0, const usfixn64 & y1, usfixn64 & q)
4  {
5          usfixn64 s0, s1, s2;
6          usfixn64 c1;
7
8          q = 0;
9
10         //      s0 = (__int128) x0 * (__int128) y0;
11         //      s0>>=64;
12         __asm__ (
13         "movq  %1, %%rax;\n\t"          // rax = a
14         "mulq  %2;\n\t"// rdx:rax = a * b
15         "movq  %%rdx, %0;\n\t"// s1 = rdx
16         : "=rm" (s0)
17         : "rm"(x0), "rm"(y0)
18         : "%rax", "%rdx");
19
20         //      s1 = (__int128) x1 * (__int128) y0;
21         //      c1 = (s1 >> 64);
22         //      s1 = s1 & (U64_MASK);
23
24         __asm__ (
25         "movq  %2, %%rax;\n\t"          // rax = a
26         "mulq  %3;\n\t"// rdx:rax = a * b
27         "movq  %%rax, %0;\n\t"// s1 = rdx
28         "movq  %%rdx, %1;\n\t"// s1 = rdx
29         : "=rm" (s1),"=rm"(c1)
30         : "rm"(x1), "rm"(y0)
31         : "%rax", "%rdx");
32
33         //      s2 = (__int128) x0 * (__int128) y1;
34         //      c2 = (s2 >> 64);
35         //      s2 = s2 & (U64_MASK);
```

```
36
37          __asm__ (
38          "movq  %2, %%rax;\n\t"          // rax = a
39          "mulq  %3;\n\t"// rdx:rax = a * b
40          "movq  %%rax, %0;\n\t"// s1 = rdx
41          //                      "movq  %%rdx, %1;\n\t"// s1 = rdx
42          "addq  %%rdx, %1;\n\t"// s1 = rdx
43          : "=rm" (s2),"=rm"(c1)
44          : "rm"(x0), "rm"(y1)
45          : "%rax", "%rdx");
46
47          //      c1+=c2;
48          q += c1;
49          //      s3 = (__int128) x1 * (__int128) y1;
50          q += x1 * y1;
51
52          __asm__ (
53          "movq  %1, %%rax;\n\t"          // rax = a
54          "addq  %2, %%rax;\n\t"// rdx:rax = a * b
55          "adcq  $0x0, %0;\n\t"
56          "addq  %3, %%rax;\n\t"// rdx:rax = a * b
57          "adcq  $0x0, %0;\n\t"
58          //                      "movq  %%rax, %0;\n\t"// s1 = rdx
59          : "+rm"(q)
60          : "rm" (s0), "rm"(s1), "rm"(s2)
61          : "%rax");
62  }
```

Listing A.1: Multiplication between two 128-bit numbers

```
1   void crt_mult_sub_u192_with_reduction(const usfixn64 &a1, const usfixn64
        &a2,
2   const crt_u192_data & data, usfixn64 &s0, usfixn64 & s1)
3   {
4           usfixn64 t[4];
5           usfixn64 q[2];
6           __int128 r[2];
7
8           mult_u64_u64(a1, data.m2, t[0], t[1]);
9           mult_u64_u64(a2, data.m1, t[2], t[3]);
10
11          mult_u128_u128_hi128(t[0], t[1], data.p1_inv_m, data.p1_inv_q, q
                [0]);
12          mult_u128_u128_hi128(t[2], t[3], data.p2_inv_m, data.p2_inv_q, q
                [1]);
13          usfixn64 m0, m1;
14
15          __asm__ (
16          "movq  %2, %%rax;\n\t"          // rax = a
17          "mulq  %3;\n\t"// rdx:rax = a * b
18          "movq  %%rax, %0;\n\t"// s1 = rdx
19          "movq  %%rdx, %1;\n\t"// s1 = rdx
20          : "=rm" (m0),"=rm"(m1)
```

```
21          : "rm"(q[0]), "rm"(data.p1)
22          : "%rax", "%rdx");
23
24          m0 = U64_MASK - m0;
25          m1 = U64_MASK - m1;
26
27          __asm__ (
28          "addq %2, %0; \n\t"
29          "adcq %3, %1; \n\t"
30          "addq $0x1, %0; \n\t"
31          "adcq $0x0, %1; \n\t"
32          : "+rm" (t[0]),"+rm"(t[1])
33          : "rm"(m0), "rm"(m1)
34          : );
35
36          ////////////////////////////////////
37
38          m0 = 0;
39          m1 = 0;
40
41          __asm__ (
42          "movq  %2, %%rax;\n\t"           // rax = a
43          "mulq  %3;\n\t"// rdx:rax = a * b
44          "movq  %%rax, %0;\n\t"// s1 = rdx
45          "movq  %%rdx, %1;\n\t"// s1 = rdx
46          : "=rm" (m0),"=rm"(m1)
47          : "rm"(q[1]), "rm"(data.p2)
48          : "%rax", "%rdx");
49
50          m0 = U64_MASK - m0;
51          m1 = U64_MASK - m1;
52
53          __asm__ (
54          "addq %2, %0; \n\t"
55          "adcq %3, %1; \n\t"
56          "addq $0x1, %0; \n\t"
57          "adcq $0x0, %1; \n\t"
58          : "+rm" (t[2]),"+rm"(t[3])
59          : "rm"(m0), "rm"(m1)
60          : );
61
62          ////////////////////////////////////
63
64          if (t[0] >= data.p1)
65          t[0] -= data.p1;
66          if (t[2] >= data.p2)
67          t[2] -= data.p2;
68
69          //      r[0] = (__int128) t[0] ;///+ ((__int128) t[1] << 64);
70          //      r[1] = (__int128) t[2] ;//+ ((__int128) t[3] << 64);
71
72          mult_u64_u64(t[0], data.p2, t[0], t[1]);
73          mult_u64_u64(t[2], data.p1, t[2], t[3]);
74
```

```
75          m0 = t[0];
76          m1 = t[1];
77          __asm__ (
78          "addq %2, %0; \n\t"
79          "adcq %3, %1; \n\t"
80          //                                          "addq $0x1, %0; \n\t"
81          //                                          "adcq $0x0, %1; \n\t"
82          : "+rm" (t[0]),"+rm"(t[1])
83          : "rm"(t[2]), "rm"(t[3])
84          : );
85          if ((t[1] > data.p1p2_q) || ((t[1] == data.p1p2_q) && (t[0] >
                data.p1p2_m)))
86          {
87                  m0 = U64_MASK - data.p1p2_m;
88                  m1 = U64_MASK - data.p1p2_q;
89                  __asm__ (
90                  "addq %2, %0; \n\t"
91                  "adcq %3, %1; \n\t"
92                  "addq $0x1, %0; \n\t"
93                  "adcq $0x0, %1; \n\t"
94                  : "+rm" (t[0]),"+rm"(t[1])
95                  : "rm"(m0), "rm"(m1)
96                  : );
97
98          }
99          s0 = t[0];
100         s1 = t[1];
101 }
```

Listing A.2: Chinese Remainder Algorithm

```
1  void __inline__ div_by_const_R(const usfixn64 x0_u64, const usfixn64
       x1_u64,
2  const usfixn64 r0, const usfixn64 r1, usfixn64 & q, usfixn64 & m)
3  {
4           //      r_inv= (u128/r_in);
5           //      r1,r0=[r_inv/u64, r_inv%u64];
6           //      x1,x0=[x/u64, x%u64];
7           //      v0=x0*r0;
8           //      v1=x0*r1;
9           //      v2=x1*r0;
10
11          __int128 v0, v1, v2, q0;
12          usfixn64 x0 = x0_u64;
13          usfixn64 x1 = x1_u64;
14          v0 = 0;
15          v1 = 0;
16          v2 = 0;
17
18          v0 = (__int128) x0 * (__int128) r0;
19          v1 = (__int128) x0 * (__int128) r1;
20          v2 = (__int128) x1 * (__int128) r0;
21
```

```
        v0 >>= 64;
        v1 += (v0);
        v2 += v1;
        v2 >>= 64;

        //      q0 = 0;
        q0 = (__int128) x1 * (__int128) r1;
        q0 += v2;

        //      m0=x-(q0*r_in);
        __int128 m0;
        m0 = (__int128) (1L << 64);
        m0 *= (__int128) x1;
        m0 += (__int128) x0;

        m0 = m0 - q0 * (__int128) (R);

        if (m0 >= (__int128) (R))
        {
                printf("carry\n");
                m0 -= (__int128) R;
                q0 += 1;
        }

        m = (usfixn64) (m0 & U64_MASK);
        q = (usfixn64) (q0 & (U64_MASK));

        if ((q0 >> 64) > 0)
        {
                printf("WARNING: q >= u64!\n");
        }
}
```

Listing A.3: Computing the quotient and remainder of a machine word size number divided by a radix r

```
void lhc_by_R_u128(const usfixn64 x0, const usfixn64 x1, const usfixn64
    & r0,
const usfixn64 & r1, const usfixn64 & u64_mod_R_q,
const usfixn64 & u64_mod_R_m, usfixn64 & s0, usfixn64 &s1, usfixn64 &s2)
{
        //def div_by_R_u128(x0,x1,x2=0,v=1):
        //
        //      usfixn64 x2 = 0;
        usfixn64 qb, mb;
        usfixn64 m0, q0;
        usfixn64 m1, q1;
        //      ### should be precomputed
        //      [mb,qb]=div_by_const_R(u64);
        //      div_by_const_R(0, 1, r0, r1, qb, mb);

        qb = u64_mod_R_q;
        mb = u64_mod_R_m;
```

```
17          //      [m0,q0]=div_by_const_R(x0);
18          div_by_const_R(x0, 0, r0, r1, q0, m0);
19
20          //      # q1=x1/R;
21          //      # m1=x1%R;
22          //      [m1,q1]=div_by_const_R(x1);
23
24          div_by_const_R(x1, 0, r0, r1, q1, m1);
25
26          __int128 l0, l1;
27          __int128 h0, h1;
28          __int128 c0, c1;
29
30          l0 = m0;
31          l1 = (__int128) m1 * (__int128) mb;
32          //      #l2=0;#x2*R0_u128;
33
34          h0 = (__int128) q0;
35          h1 = (__int128) q1 * (__int128) mb + (__int128) qb * (__int128)
                m1;
36          //      #h2=0;#x2*R1_u128;
37
38          c0 = 0;
39          c1 = (__int128) q1 * (__int128) qb;
40          //      #c2=0;#x2*R2_u128
41
42
43          usfixn64 lhc_l0h0c0[3] =
44          { l0, h0, c0 };
45          //      lhc_l0h0c0=[l0,h0,c0];
46
47          usfixn64 lhc_l1c1[3] =
48          { 0, 0, c1 };
49          usfixn64 lhc_l2c2[3] =
50          { 0, 0, 0 };
51          usfixn64 lhc_h1h2[3] =
52          { 0, 0, 0 };
53          usfixn64 lhc_ans[3] =
54          { 0, 0, 0 };
55
56          div_by_const_R(l1 & U64_MASK, l1 >> 64, r0, r1, lhc_l1c1[1],
                lhc_l1c1[0]);
57          div_by_const_R(h1 & U64_MASK, h1 >> 64, r0, r1, lhc_h1h2[2],
                lhc_h1h2[1]);
58
59          //      lhc_ans=add_lhc(lhc_l0h0c0,lhc_l1c1)
60          //      lhc_ans=add_lhc(lhc_ans,lhc_l2c2)
61
62          add_lhc(lhc_l0h0c0, lhc_l1c1, lhc_ans);
63          add_lhc(lhc_ans, lhc_h1h2, lhc_ans);
64
65          //      printf("l=%lu, h=%lu, c=%lu\n", lhc_ans[0], lhc_ans[1],
                lhc_ans[2]);
66
```

```
67          s0 = lhc_ans[0];
68          s1 = lhc_ans[1];
69          s2 = lhc_ans[2];
70
71  }
```

Listing A.4: (l,h,c) Algorithm

```
1   inline sfixn MontMulModSpe_OPT3_AS_GENE_INLINE(sfixn a,sfixn b,sfixn
        MY_PRIME, sfixn INV_PRIME){
2           asm("mulq %2\n\t"
3           "movq %%rax,%%rsi\n\t"
4           "movq %%rdx,%%rdi\n\t"
5           "imulq %3,%%rax\n\t"
6           "mulq %4\n\t"
7           "add %%rsi,%%rax\n\t"
8           "adc %%rdi,%%rdx\n\t"
9           "subq %4,%%rdx\n\t"
10          "mov %%rdx,%%rax\n\t"
11          "sar $63,%%rax\n\t"
12          "andq %4,%%rax\n\t"
13          "addq %%rax,%%rdx\n\t"
14          : "=d" (a)
15          : "a"(a),"rm"(b),"b"((sfixn) INV_PRIME),"c"((sfixn) MY_PRIME)
16          :"rsi","rdi");
17          return a;
18  }
```

Listing A.5: Montgomery multiplication for 64-bit numbers

Curriculum Vitae

Name:	Linxiao Wang
Post-Secondary Education and Degrees:	Renmin University of China Beijing, China 2011 - 2015 B.Eng. University of Western Ontario London, ON 2016 - 2018 M.Sc.
Related Work Experience:	Teaching Assistant University of Western Ontario 2017 - 2018 Research Assistant University of Western Ontario 2016 - 2018

Publications:

1. Qin, B., Wang, L., Wang, Y., Wu, Q., Shi, W., & Liang, B. (2014, October). Efficient sub-/inter-group key distribution for ad hoc networks. In International Conference on Network and System Security (pp. 448-461). Springer, Cham.
2. Qin, B., Wang, L., Wang, Y., Wu, Q., Shi, W., & Liang, B. (2016). Versatile lightweight key distribution for big data privacy in vehicular ad hoc networks. Concurrency and Computation: Practice and Experience, 28(10), 2920-2939.